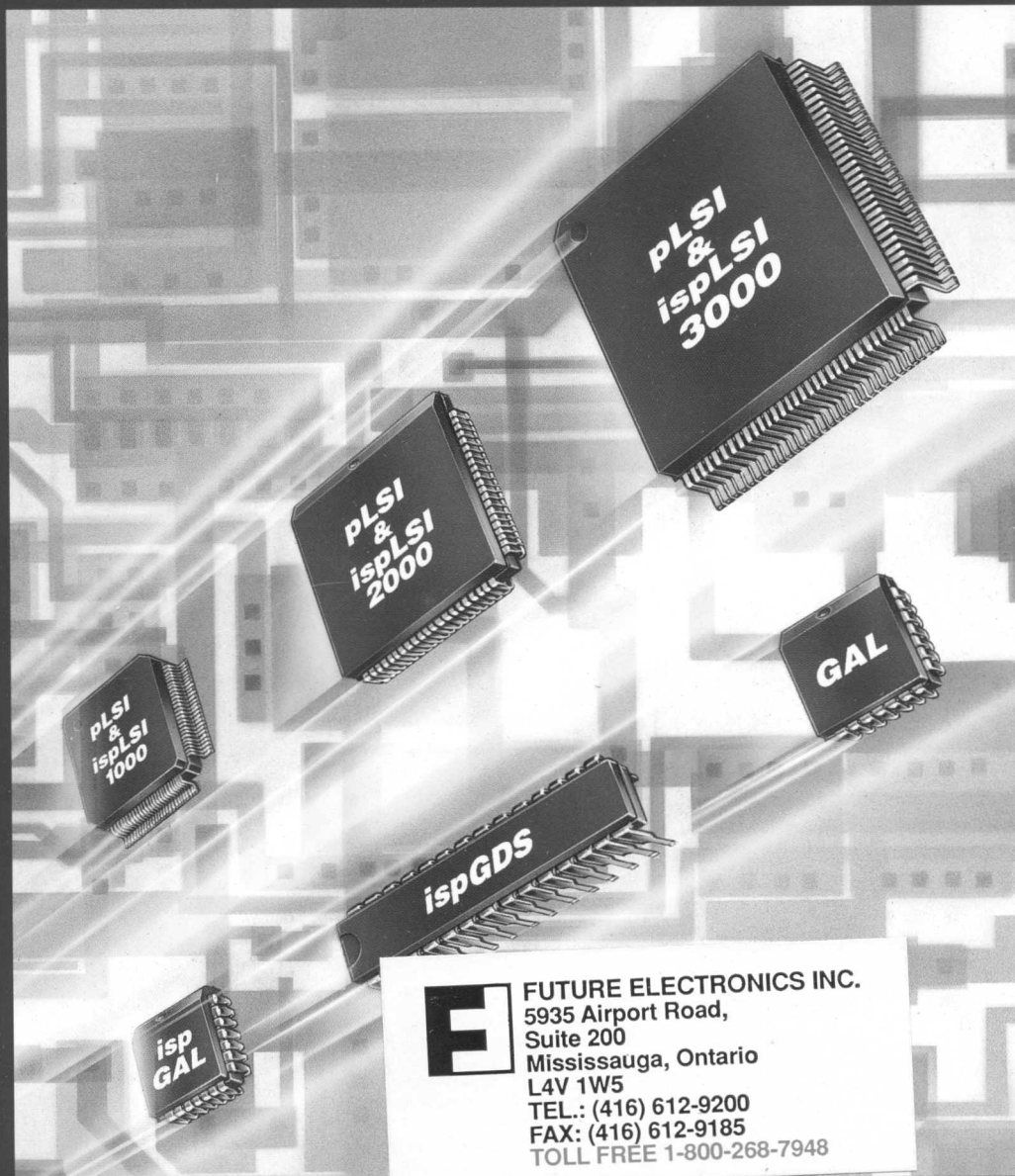


Lattice Handbook

1994



PLSI & ispLSI 3000

PLSI & ispLSI 2000

PLSI & ispLSI 1000

GAL

ispGDS

isp GAL

F **FUTURE ELECTRONICS INC.**
5935 Airport Road,
Suite 200
Mississauga, Ontario
L4V 1W5
TEL.: (416) 612-9200
FAX: (416) 612-9185
TOLL FREE 1-800-268-7948

 **Lattice™**

Lattice Handbook

1994



Copyright © 1994 Lattice Semiconductor Corporation

FROMOS, GAL, LEGAL, POS, PL81, SHOOT FORM, and UPMOS are registered trademarks of Lattice Semiconductor Corporation. GEMMA Array Logic, LSP, LOGCODE, LOGDOWNLOAD, LOGGDS, LOGLSH, LOGSTREAM, LATCH-LOCK, POS+ and RPT are trademarks of Lattice Semiconductor Corporation.

All brand names or product names mentioned are trademarks or registered trademarks of their respective holders.

Lattice Semiconductor Corporation products are made under one or more of the following U.S. and international patents: 4,781,788 US, 4,786,888 US, 4,882,848 US, 4,922,884 US, 4,929,888 US, 4,987,239 US, 4,996,296 US, 5,180,274 US, 5,188,198 US, 5,182,879 US, 5,197,243 US, 5,204,888 US, 5,281,318 US, 5,287,318 US, 5,307,218 US, 5,348,228 US, 5,351,189 US, 5,196,091 EP, 0,166,771 EP, 0,166,771 UK, 0,166,771 WD.



LATTICE SEMICONDUCTOR CORP.
2555 Northeast Mission Court
Hillsboro, Oregon 97124 U.S.A.
Tel: (503) 651-0118
Fax: (503) 651-3037



Copyright © 1994 Lattice Semiconductor Corporation

E²CMOS, GAL, ispGAL, pDS, pLSI, Silicon Forest and UltraMOS are registered trademarks of Lattice Semiconductor Corporation. Generic Array Logic, ISP, ispCODE, ispDOWNLOAD, ispGDS, ispLSI, ispSTREAM, Latch-Lock, pDS+ and RFT are trademarks of Lattice Semiconductor Corporation

All brand names or product names mentioned are trademarks or registered trademarks of their respective holders.

Lattice Semiconductor Corporation products are made under one or more of the following U.S. and international patents: 4,761,768 US, 4,766,569 US, 4,833,646 US, 4,852,044 US, 4,855,954 US, 4,879,688 US, 4,887,239 US, 4,896,296 US, 5,130,574 US, 5,138,198 US, 5,162,679 US, 5,191,243 US, 5,204,556 US, 5,231,315 US, 5,231,316 US, 5,237,218 US, 5,245,226 US, 5,251,169 US, 0194091 EP, 0196771B1 EP, 0196771 UK, 0196771 WG.

LATTICE SEMICONDUCTOR CORP.
5555 Northeast Moore Court
Hillsboro, Oregon 97124 U.S.A.
Tel.: (503) 681-0118
FAX: (503) 681-3037

How To Use This Handbook

Background

Lattice Semiconductor Corporation, founded in 1983 and based in Hillsboro, Oregon, for over a decade has been providing innovative solutions to the manufacturers of high performance systems. Lattice pioneered non-volatile, reprogrammable logic with its UltraMOS E²CMOS technology. This technology, combined with the Lattice GAL architectures, have established Lattice products as the industry standard in low density programmable logic. Lattice's ispLSI and pLSI families of high density PLDs combine leadership performance and density with in-system programmability to establish the high-density programmable logic standard of the 1990's.

What This Handbook Contains

This handbook offers product overviews, architecture overviews, applications notes, and various other pieces of information about Lattice's programmable devices and development tools. Please consult the latest Lattice Data Book for more detailed information on device and software specifications.

Additional Information

For information on product availability and pricing, please contact your Lattice Sales Representative or Distributor. A listing of all Lattice Sales Offices, Sales Representatives, and Distributors is at the end of this handbook.

For immediate help with technical questions or access to selected applications described inside, please call:

Applications Hotline

GAL Products: Tel. 1-800-FASTGAL (327-8425), FAX (503) 681-3037
ispLSI and pLSI Products: Tel. 1-800-LATTICE (528-8423), FAX (408) 944-8450

Electronic Bulletin Board

GAL Products: (503) 693-0215
ispLSI and pLSI Products: (408) 980-9814

Acknowledgments
We thank those dedicated employees whose hard work and long hours have made Lattice products and this book a reality.

How To Use This Handbook

Background

Lattice Semiconductor Corporation, founded in 1983 and based in Hillsboro, Oregon, for over a decade has been providing innovative solutions to the manufacturers of high performance systems. Lattice pioneered non-volatile, reprogrammable logic with its UltraMOS E²CMOS technology. This technology, combined with the Lattice GAL architectures, have established Lattice products as the industry standard in low density programmable logic. Lattice's ispLSI and pLSI families of high density FPLDs combine leading performance and density with in-system programmability to establish the high-density, programmable logic standard of the 1990's.

What This Handbook Contains

This handbook offers product overviews, architecture overviews, applications notes, and various other pieces of information about Lattice's programmable devices and development tools. Please consult the latest Lattice Data Book for more detailed information on devices and software specifications.

Additional Information

For information on product availability and pricing, please contact your Lattice Sales Representative or Distributor. A listing of all Lattice Sales Offices, Sales Representatives, and Distributors is at the end of this handbook.

For immediate help with technical questions or access to selected applications described inside, please call:

Applications Hotline

ispLSI and pLSI Products: Tel: 1-800-LATTICE (528-8423), FAX: (408) 884-8420
GAL Products: Tel: 1-800-FASTGAL (327-8425), FAX: (503) 681-3037

Electronic Bulletin Board

ispLSI and pLSI Products: (408) 880-0814
GAL Products: (503) 683-0215

Acknowledgments

We thank those dedicated employees whose hard work and long hours have made Lattice products, and this book, a reality.

Table of Contents

Section 1: Introduction

Introduction	1-1
--------------------	-----

Section 2: ispLSI and pLSI Architecture Overview

Introduction to ispLSI and pLSI Families	2-1
1000 Family Architecture Description	2-11
2000 Family Architecture Description	2-25
3000 Family Architecture Description	2-31
ispLSI Architecture and Programming	2-39

Section 3: ispLSI and pLSI Development Tools

Lattice Design Tool Strategy	3-1
System Design Process	3-3
ispLSI and pLSI Design Flow	3-5

Section 4: ispLSI and pLSI Application Notes

Selecting the Right High Density Device	4-1
Beginner's Guide to ispLSI and pLSI	4-7
ispLSI and pLSI: A Multiple Function Solution	4-21
Programming Multiple ISP Devices: Daisy Chain Configuration	4-41
Compiling Multiple PLDs into ispLSI and pLSI Devices	4-47
Adders/Subtractors in pLSI	4-55
Crosspoint Switch Implementation Using the pLSI 1032	4-61
Building Modulo N Counters Using ispLSI and pLSI Devices	4-69
Phase Locked Loops (PLL) in High Speed Designs	4-71
Video Graphics Controller	4-75
A Digital Clock Design Example	4-95
ispLSI Configurable Memory Controller	4-105
Bar Code Reader	4-121
High Density PLD Solutions for High Speed RISC/CISC Systems	4-139
SCSI Interface with the ispLSI 3256	4-145
PCI Bus Implementation	4-155
Programming ispLSI Devices with a Tester	4-179

Section 5: GAL Architecture Overview

Introduction to Generic Array Logic	5-1
---	-----

Section 6: GAL Development Tools

Using GAL Development Tools	6-1
GAL Development Support	6-11
Copying PAL, EPLD and PEEL Patterns into GAL Devices	6-13

Section 7: GAL Application Notes

Zero-Power GAL Devices	7-1
The GAL16VP8 and GAL20VP8	7-5
The GAL18V10 Advantage	7-7

GAL20RA10: Programmable Clocks Improve System Performance	7-11
GAL6002 Designs Using ABEL and CUPL	7-13
GAL6002: 4-to-1 RS232 Port Multiplexer	7-17
VME Bus Arbitration Using a GAL22V10	7-21
GAL16VP8/20VP8: Bus Arbitration Circuit	7-25
GAL20XV10: Data Block Transfer Address Detector	7-29
GAL26CV12: Programmable Frequency Divider	7-33
Section 8: In-System Programmable Generic Digital Switch (ispGDS)	
Introduction to the ispGDS Family	8-1
Using ispGDS Devices	8-5
ispGDS Compiler Support	8-7
Lattice's Solution for Plug-and-Play	8-9
Section 9: Design Techniques	
User Electronic Signature	9-1
Driving CMOS Inputs with GAL Devices	9-3
Metastability Report	9-5
Latch-Up Protection	9-19
Section 10: Article Reprints	
Selecting the Best Device for In-System Programmability	10-1
Enhanced E ² PLDs Hit Speed and Density Highs	10-7
Complex State Machine Design with Complex PLDs	10-11
Avoid the Pitfalls of Hi Speed Logic Design	10-16
PLD-Design Methods Migrate Existing Designs to High-Capacity Devices	10-23
In-System Programmable Logic in High Volume Manufacturing	10-30
A Token Ring Network Adapter Card	10-37
A Decision Process Used for FPGA Selection in	
Digital Signal Processing for Fiber Optic Sensors	10-44
Learn the Fundamentals of Digital Filter Design	10-51
State Machine Design for High Speed PowerPC RISC Microprocessor Systems	10-59
Applying In-System Reprogrammability in a REFLECTIVE MEMORY Bus Controller	10-66
PLD Usage Generalizes HDTV Frame Buffer Interface	10-73
Section 11: Technology, Quality, and Reliability Overview	
Quality Assurance Program	11-1
Qualification Program	11-3
E ² CMOS Testability Improves Quality	11-5
Technology and Reliability	11-7
ISO 9000 Program	11-11
Section 12: General Section	
Lattice Bulletin Board Systems	12-1
Cost of Ownership: An Overview	12-7
Hidden Costs in PLD Usage	12-9
ISP: Winning at the Bottom Line	12-15
Gate Array and High Density PLD Cost Analysis	12-17
Sales Offices	12-19

Section 1: Introduction

Introduction 1-1

Section 2: ispLSI and pLSI Architecture Overview**Section 3: ispLSI and pLSI Development Tools****Section 4: ispLSI and pLSI Application Notes****Section 5: GAL Architecture Overview****Section 6: GAL Development Tools****Section 7: GAL Application Notes****Section 8: In-System Programmable Generic Digital Switch (ispGDS)****Section 9: Design Techniques****Section 10: Article Reprints****Section 11: Technology, Quality, and Reliability Overview****Section 12: General Section**



Section 1: Introduction	1-1
Section 2: iapLSI and pLSI Architecture Overview	
Section 3: iapLSI and pLSI Development Tools	
Section 4: iapLSI and pLSI Application Notes	
Section 5: GAL Architecture Overview	
Section 6: GAL Development Tools	
Section 7: GAL Application Notes	
Section 8: In-System Programmable General Digital Switch (pGDS)	
Section 9: Design Techniques	
Section 10: Article Reprints	
Section 11: Technology, Quality, and Reliability Overview	
Section 12: General Section	

Background

Through pioneering efforts in applying E²CMOS[®] technology to programmable logic, Lattice has established the GAL[®] family of products as the industry standard worldwide. With the introduction of the high-density programmable Large Scale Integration (pLSI[®]) devices and in-system programmable Large Scale Integration (ispLSI[™]) devices, Lattice has become the world's largest supplier of low-density CMOS PLDs and the fastest growing supplier of high-density CMOS PLDs.

Lattice has recently introduced two new low-density in-system programmable devices: the ispGAL22V10 and ispGDS[™]. The ispGAL22V10 brings on-the-fly system logic reconfigurability to the industry standard GAL22V10. The ispGDS (in-system programmable Generic Digital Switch) family further extends Lattice's programmable technology beyond logic to board interconnect and signal routing. The ispGDS family opens new possibilities for system designers and is just the first of a series of application specific programmable solutions that will be provided by Lattice in the future.

The Lattice Advantage

Time-to-Market

E²CMOS PLDs enable system designers to meet ever-shrinking time-to-market constraints while avoiding the significant development costs, lead times and dedicated inventories associated with traditional ASIC and bipolar PLD solutions.

Flexibility

Programmable and reprogrammable devices enable fast and easy modifications to system designs.

Product Differentiation

Lattice's programmable devices allow design engineers to easily differentiate their end-product through proprietary feature enhancements. This is particularly true when a system utilizes the non-volatile ISP[™] (In-System Programmable) technology pioneered by Lattice.

Inventory Reduction

A single standard part type can be used in multiple, diverse applications. Just five GAL architectures replace virtually all bipolar PAL[®] architectures (see figure 1).

Products

The Lattice PLD product offering can be segmented into two strategic product thrusts:

Low Density: GAL Family

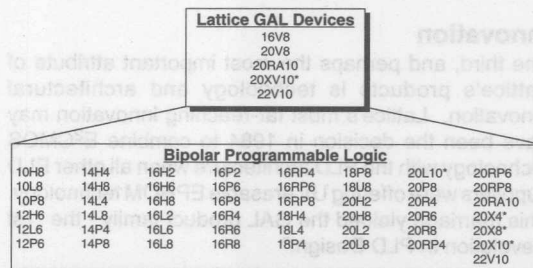
- 100 - 1,000 Gates
- The Highest Performance PLDs from any Supplier
- Superior Replacements for Bipolar and CMOS PLD Architectures
- E²CMOS Low-Power, Quality and Reliability
- Broadest Range of PLD Architectures Offering Features not Available in other PLDs
- Pioneering Non-volatile In-System Programmability (ISP)

High Density: ispLSI and pLSI Families

- 1,000 - 14,000 Gates (World's Largest)
- World's Fastest High-Density PLDs (HDPLDs)
- Superior HDPLD Architecture (Flexible, Predictable Performance)
- Pioneering Non-volatile In-System Programmability (ISP)
- Range of Effective Development Tool Options

Lattice Product Features

Figure 1. Five GAL devices replace virtually all bipolar PAL devices.



* GAL20XV10 replaces 20L10, 20X10, 20X8 and 20X4

Introduction

There are three fundamental features which Lattice PLDs share: E²CMOS technology, performance leadership and innovation.

E²CMOS Technology

All GAL, pLSI and ispLSI devices are manufactured using Lattice's proprietary high-speed UltraMOS® E²CMOS technology. Lattice is unique among "fab-less" companies in that the process technology development is actually done by Lattice. UltraMOS technology successfully combines the best features of CMOS and NMOS process technology to yield PLDs with the following key features:

- Industry Leading Performance
- High Logic Densities
- Low Power Consumption
- Non-Volatile, In-System Programmability
- Fast Erase and Reprogram Times
- 100% Full Parametric Testability
- 100% Programming and Functional Yields

Performance Leadership

Lattice continues its long track record of producing the fastest CMOS PLDs in the market. These industry-leading high-performance products are typically available to the market months ahead of any other PLD supplier. As a result, Lattice customers have always been able to take full advantage of next generation microprocessor speeds and bring out industry leading end-products of their own, thus fueling their own success.

While speed continues to be a top priority, Lattice has also introduced PLD families which address other logic design concerns such as low power ("Zero-Power" GAL16/20V8Z and GAL16/20V8ZD), high output drive (GAL16/20VP8) and logic density (GAL26CV12).

Innovation

The third, and perhaps the most important attribute of Lattice's products is technology and architectural innovation. Lattice's most far-reaching innovation may have been the decision in 1984 to combine E²CMOS technology with the PLD architecture when all other PLD suppliers were offering UV erasable EPROM technology. This marriage yielded the GAL product family - the "1st Revolution in PLD Design."

Lattice innovation also started the "2nd Revolution in PLD Design" with the introduction of the first non-volatile in-system programmable high-density PLD family — ispLSI and reinforced with the introduction of the ispGAL22V10 and ispGDS families.

The ISP concept, and the ispLSI, ispGAL and ispGDS families in particular, dramatically impact system development and manufacturing. Lattice ISP solutions deliver:

Effortless Prototyping: Design iterations can be downloaded directly to the ISP device soldered onto the prototype board.

Reconfigurable Systems: A single generic board can be "personalized" to one of many system configurations at final board-level test.

Simplified Manufacturing: Eliminates all stand-alone programming steps. Device programming can be done as part of board-level testing. The result is no misprogrammed devices, no inventory headaches keeping track of patterned devices, and no PLD rework costs.

No More Bent Leads: ISP technology also solves the handling problems associated with high pin count, fine pitch packages (PQFP, TQFP etc.). Programming devices in-system eliminates bent leads and unreliable solder joints.

Summary

Lattice, the leader in E²CMOS PLDs, is committed to providing its customers with industry-leading programmable solutions. We realize that your system design requirements and time-to-market pressures will only get tougher in the future. Lattice is committed to supporting you with state-of-the-art products with the performance, architecture, quality and reliability that satisfy your requirements.

Section 1: Introduction**Section 2: ispLSI and pLSI Architecture Overview**

Introduction to ispLSI and pLSI Families	2-1
1000 Family Architecture Description	2-11
2000 Family Architecture Description	2-25
3000 Family Architecture Description	2-31
ispLSI Architecture and Programming	2-39

Section 3: ispLSI and pLSI Development Tools**Section 4: ispLSI and pLSI Application Notes****Section 5: GAL Architecture Overview****Section 6: GAL Development Tools****Section 7: GAL Application Notes****Section 8: In-System Programmable Generic Digital Switch (ispGDS)****Section 9: Design Techniques****Section 10: Article Reprints****Section 11: Technology, Quality, and Reliability Overview****Section 12: General Section**

Section 1: Introduction	
Section 2: iqlsi and qlsi Architecture Overview	
Introduction to iqlsi and qlsi Families	2-1
1000 Family Architecture Description	2-11
2000 Family Architecture Description	2-25
3000 Family Architecture Description	2-31
iqlsi Architecture and Programming	2-39
Section 3: iqlsi and qlsi Development Tools	
Section 4: iqlsi and qlsi Application Notes	
Section 5: GAL Architecture Overview	
Section 6: GAL Development Tools	
Section 7: GAL Application Notes	
Section 8: In-System Programmable Generic Digital Switch (ispqds)	
Section 9: Design Techniques	
Section 10: Article Reprints	
Section 11: Technology, Quality, and Reliability Overview	
Section 12: General Section	

Introduction to ispLSI™ and pLSI® Families

The Lattice ispLSI and pLSI Families

The Lattice programmable Large Scale Integration (pLSI) and in-system programmable Large Scale Integration (ispLSI) families are the logical choice for your next design project. They're the first programmable logic devices to combine the performance and ease of use of PLDs with the density and flexibility of FPGAs. And at 135 MHz system speed, and up to 14000 PLD gates, they're the world's fastest and highest density programmable logic devices!

There are three ispLSI and pLSI families to fit your specific application needs. Lattice's premier ispLSI and pLSI 1000 family implements high integration functions such as controllers, LANs and encoders at high speeds. The high performance ispLSI and pLSI 2000 family with its large number of I/Os handles timers, counters as well as timing critical interfaces to high speed RISC/CISC microprocessors. The highest density ispLSI and pLSI 3000 family integrates complete system logic, DSP functions, and entire encryption or compression logic into a single package, while delivering superior performance.

The ispLSI 1000, 2000 and 3000 families pioneer non-volatile, in-system programmability, a technology that allows real-time programming, less expensive manufacturing and end-user system reconfiguration.

All the development tools you need are available from Lattice - tools ranging from Lattice's own entry level software to higher level, third-party design environments. With these tools, you'll be completing your circuit designs in hours instead of weeks or months.

ispLSI and pLSI 1000: The Premier High Density Family

- 110 MHz system performance
- 10 ns pin-to-pin delay (maximum)
- 2000-8000 PLD gates
- 44-pin to 128-pin packages

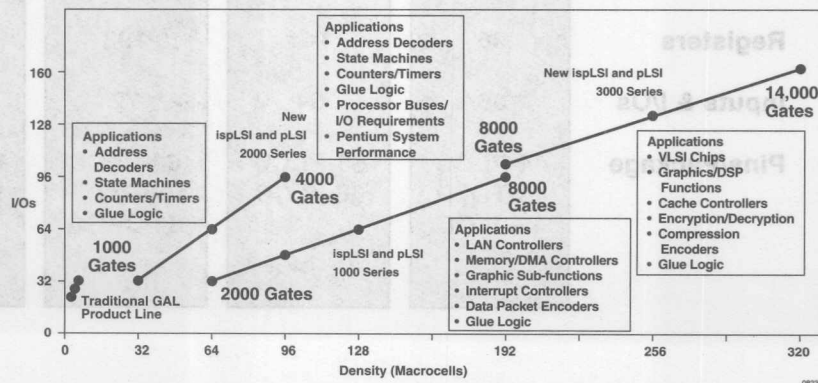
ispLSI and pLSI 2000: Unparalleled System Performance

- 135 MHz system performance (world's fastest!)
- 7.5 ns pin-to-pin delay (maximum)
- 1000-4000 PLD gates
- 44-pin to 128-pin packages
- High I/O to Logic Ratio

ispLSI and pLSI 3000: Density with Performance

- 110 MHz system performance
- 10 ns pin-to-pin delay (maximum)
- 8000-14000 PLD gates (world's largest!)
- 128-pin to 208-pin packages
- Boundary scan for enhanced testability (IEEE 1149.1)

Lattice's ispLSI and pLSI Families



Introduction to ispLSI and pLSI

Family Overview

From registers to counters, from multiplexers to complex state machines, these families of high-density programmable logic will address your high-performance system logic needs.

With PLD gate densities ranging from 1,000 to 14,000, the ispLSI and pLSI devices provide the range of programmable logic solutions you need to meet design requirements today and tomorrow.

Each device contains multiple Generic Logic Blocks (GLBs), architected to maximize system flexibility and performance. And a generous supply of registers and I/O cells provides the optimum balance of internal logic and external connections. A global interconnect scheme ties everything together, enabling high logic utilization.

Table 1. ispLSI and pLSI Family Attributes

ispLSI and pLSI 1000

	ispLSI 1016	ispLSI 1024	ispLSI 1032	ispLSI 1048/1048C
Density (PLD Gates)	2000	4000	6000	8000
Speed: Fmax (MHz)	110	90	90	80
Speed: Tpd (ns)	10	12	12	15
Macrocells	64	96	128	192
Registers	96	144	192	288
Inputs & I/Os	36	54	72	106/110
Pins/Package	44-PLCC 44-TQFP 44-JLCC	68-PLCC 68-JLCC	84-PLCC 100-TQFP 84-CPGA	120-PQFP 128-PQFP

1/2/3000-2A

Introduction to ispLSI and pLSI

ispLSI and pLSI Architecture

The ispLSI and pLSI architecture was constructed with real system design requirements in mind. Figure 1 shows the representation of the pLSI 3256 architecture. This architecture provides the designer with the following advantages.

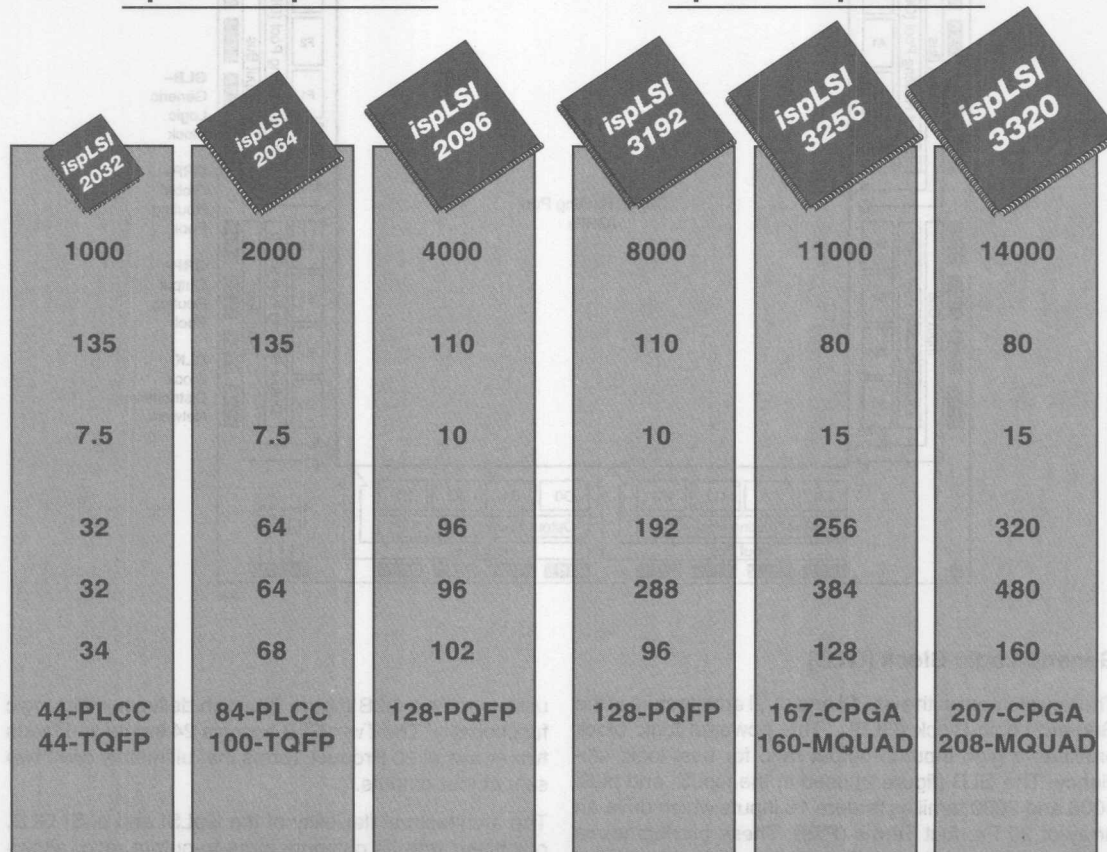
- ☐ High Speed
- ☐ Predictable performance

- ☐ Low power
- ☐ Flexible architecture
- ☐ Easy to use
- ☐ Design portability across all the families
- ☐ Non-volatile in-system programmable (ispLSI)
- ☐ Advanced Global Clock Network
- ☐ Boundary Scan (3000 Family)

2

ispLSI and pLSI 2000

ispLSI and pLSI 3000



1/2/3000-3A

Introduction to ispLSI and pLSI

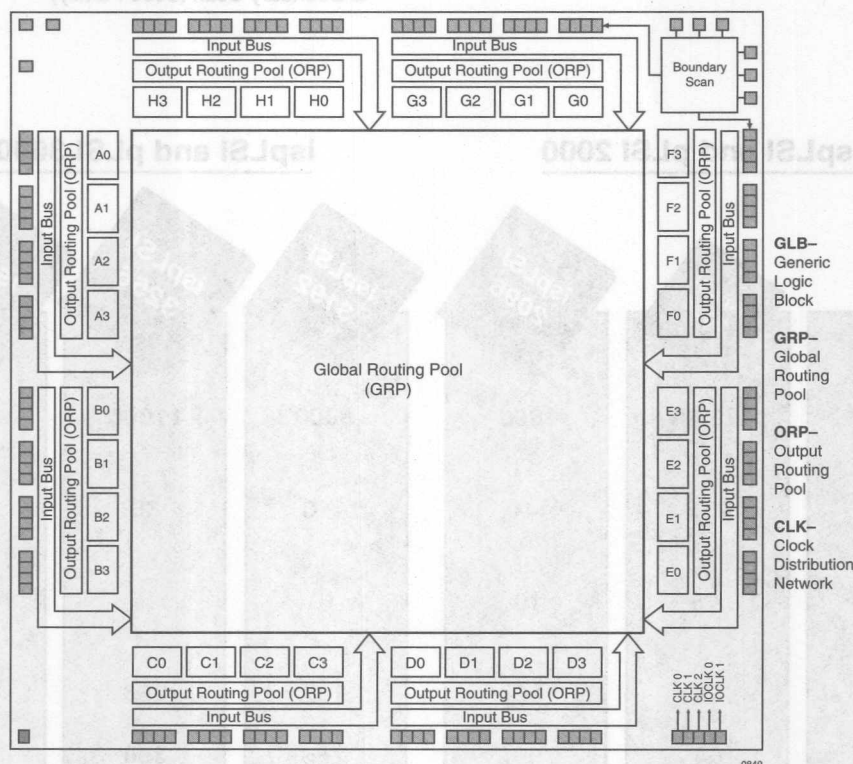
The Global Routing Pool

Central to the ispLSI and pLSI architecture is the Global Routing Pool (GRP), which connects all of the internal logic and makes it available to the designer. The GRP provides complete interconnectivity with fixed and predictable delays. This unique interconnect scheme consistently provides high performance and allows effortless implementation of complex designs.

The Output Routing Pool (ORP)

The Output Routing Pool (ORP) is a unique ispLSI and pLSI architectural feature which provides flexible connections between the GLB outputs and the output pins. This flexibility allows for "last minute" logic design changes to be implemented without changing the external pin-out.

Figure 1. pLSI 3256 Functional Block Diagram



Generic Logic Block (GLB)

The key element in the ispLSI and pLSI architecture is the Generic Logic Block (GLB). This powerful logic block provides a high input-to-output ratio for best logic efficiency. The GLB (figure 2) used in the ispLSI and pLSI 1000 and 2000 families feature 18 inputs which drive an array of 20 Product Terms (PTs). These product terms feed four outputs which effectively handle both wide and narrow gating functions. The ispLSI and pLSI 3000 family

utilizes a Twin GLB (figure 3) which delivers wider logic functionality. The Twin GLB accepts 24 inputs and feeds two arrays of 20 Product Terms that ultimately drive two sets of four outputs.

The architectural flexibility of the ispLSI and pLSI GLB, combined with its optimum input-to-output ratio, allows the GLB to implement virtually all 4-bit and 8-bit MSI functions.

An additional element of architectural flexibility is the Product Term Sharing Array (PTSA). The PTSA allows the 20 PTs from the AND array to be shared with any and all of the four GLB outputs. This ability to share PTs between all of the four GLB outputs provides a highly efficient means to implement complex state machines by eliminating duplicate product term groups.

Each of the four outputs from the PTSA feeds into a flexible Output Logic Macrocell (OLMC), consisting of a D-type flip-flop with an Exclusive-OR gate on the input.

The OLMC allows each GLB output to be configured as either combinatorial or registered. Combinatorial mode is available as AND-OR or Exclusive-OR. Registered mode is available as D, T or J-K.

The power of the GLB is further enhanced by a flexible clock distribution network. This network provides a choice of clock signals to each GLB: global synchronous clock signals or internally generated asynchronous product term clock signals.

Figure 2. ispLSI and pLSI 1000 and 2000 Family GLB

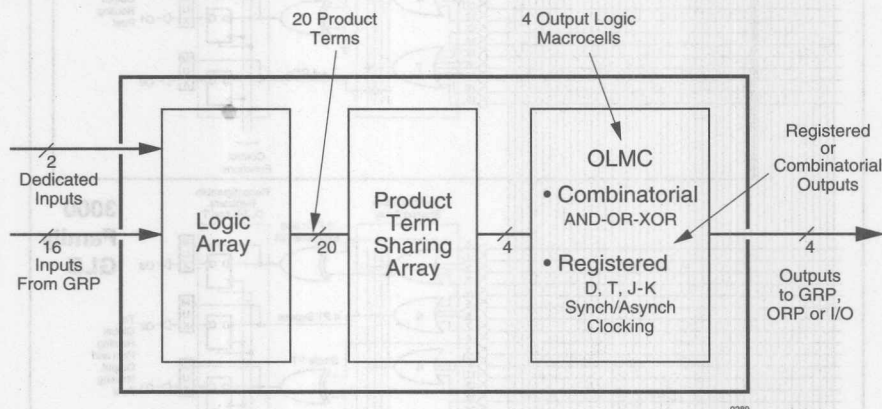
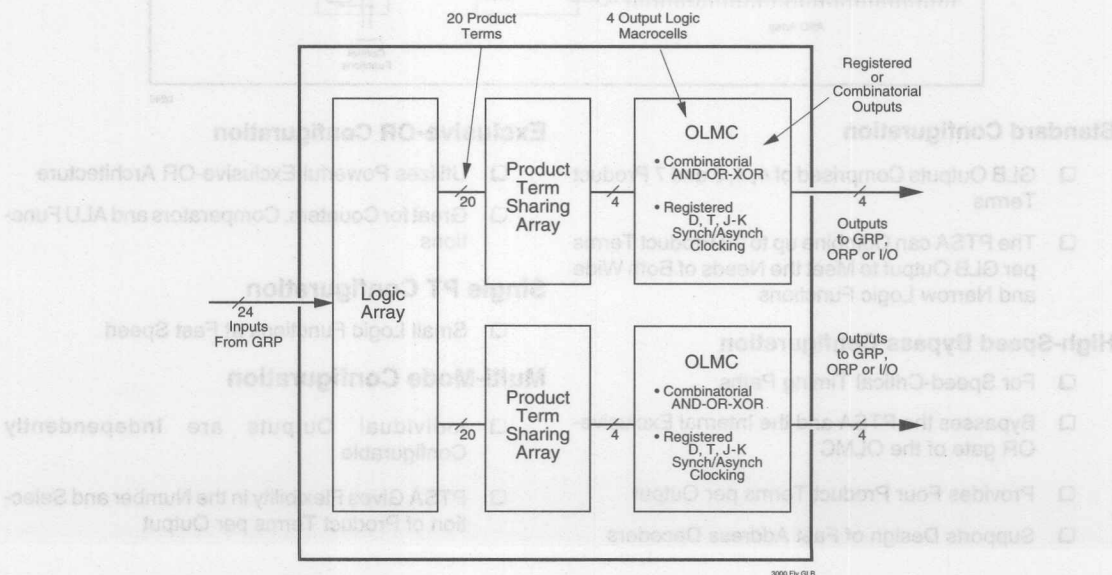
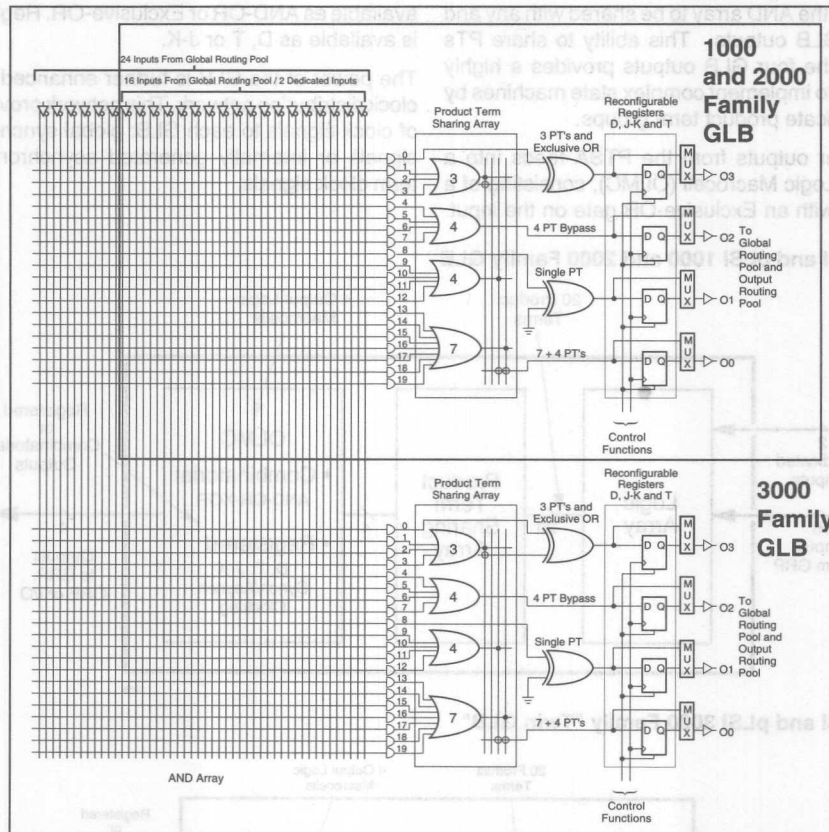


Figure 3. ispLSI and pLSI 3000 Family "Twin GLB"



Introduction to ispLSI and pLSI

Figure 4. GLB: Multi-Mode Configuration



0845

Standard Configuration

- ☐ GLB Outputs Comprised of 4, 4, 5 and 7 Product Terms
- ☐ The PTSA can Combine up to 20 Product Terms per GLB Output to Meet the Needs of Both Wide and Narrow Logic Functions

High-Speed Bypass Configuration

- ☐ For Speed-Critical Timing Paths
- ☐ Bypasses the PTSA and the Internal Exclusive-OR gate of the OLMC
- ☐ Provides Four Product Terms per Output
- ☐ Supports Design of Fast Address Decoders

Exclusive-OR Configuration

- ☐ Utilizes Powerful Exclusive-OR Architecture
- ☐ Great for Counters, Comparators and ALU Functions

Single PT Configuration

- ☐ Small Logic Functions at Fast Speed

Multi-Mode Configuration

- ☐ Individual Outputs are Independently Configurable
- ☐ PTSA Gives Flexibility in the Number and Selection of Product Terms per Output

Security Cell

A security cell is provided in the ispLSI and pLSI devices to prevent unauthorized copying of the array patterns. Once programmed, this cell prevents further read access to the functional bits in the device. This cell can only be erased by reprogramming the device, so the original configuration can never be examined once this cell is programmed.

Device Programming

ispLSI and pLSI devices can be programmed using a Lattice-approved device programmer, available from a number of third party manufacturers. Complete programming of the device takes only a few seconds. Erasing of the device is automatic and is completely transparent to the user. In-system programming is also available with ispLSI devices which allows programming on the circuit board using Lattice programming algorithms and standard 5V system power.

Latch-up Protection

ispLSI and pLSI devices are designed with an on-board charge pump to negatively bias the substrate. The negative bias is of sufficient magnitude to prevent input undershoots from causing the internal circuitry to latch-up. Additionally, outputs are designed with n-channel pull-ups instead of the traditional p-channel pull-ups to eliminate any possibility of SCR induced latching.

In-System Programmability

Lattice's ispLSI devices (in-system programmable) are the industry's only high-density programmable logic family offering non-volatile, in-system reconfigurability.

ispLSI devices are available in all three families: 1000, 2000 and 3000. The ispLSI devices are 100 percent functionally and parametrically compatible with their pLSI counterparts, with the added capability for 5-volt in-system programmability and reprogrammability.

Complex logic functions can be implemented in multiple ispLSI devices with complete on-board configurability. In-system programming of a multiple ispLSI chip solution is easily achieved through a proprietary in-system erase/program/verify technique.

In-system programmability can revolutionize the way you design, manufacture and service systems.

Prototype Board Designs

In-system programming allows you to program and modify your logic designs "in-system" without removing the device(s) from the board. This accelerates the system and board-level debug process and enables you to define the board layout earlier in the design process.

Fine Pitch Package Handling

When programming traditional PLDs, manual handling is required during both design/debugging and manufacturing stages. When using PQFPs or TQFPs, fragile leads as thin as 0.5 mm can easily bend in the programmer socket causing coplanarity damage. With ispLSI, you can solder these packages onto your printed circuit board and still program and reprogram the devices during debugging and manufacturing – without ever losing a single part due to bent leads.

Reconfigurable Systems

Your options become boundless when you have the ability to change the functionality of devices already soldered on a p.c. board. You can now implement multiple hardware configurations with the same circuit board design. A variety of protocols or system interfaces can be implemented on a generic board as the last step in the manufacturing flow.

Easier Field Updates

With software reconfigurable systems, field updates are as easy as loading a new configuration from a floppy or downloading it through a modem.

Enhanced Manufacturing Flow with ispLSI

Perhaps the most exciting benefit of the ispLSI family is its potential to streamline the manufacturing process by eliminating the separate programming and labeling steps usually associated with PLDs. Quality is enhanced when product handling steps are reduced, in this case, those associated with programming, labeling and re-inventorying multiple device types. Eliminating socketing further improves quality and reduces board cost. Figure 6 shows the enhanced manufacturing with the ispLSI device.

Introduction to ispLSI and pLSI

Figure 5. In-System Programmable Graphics Board

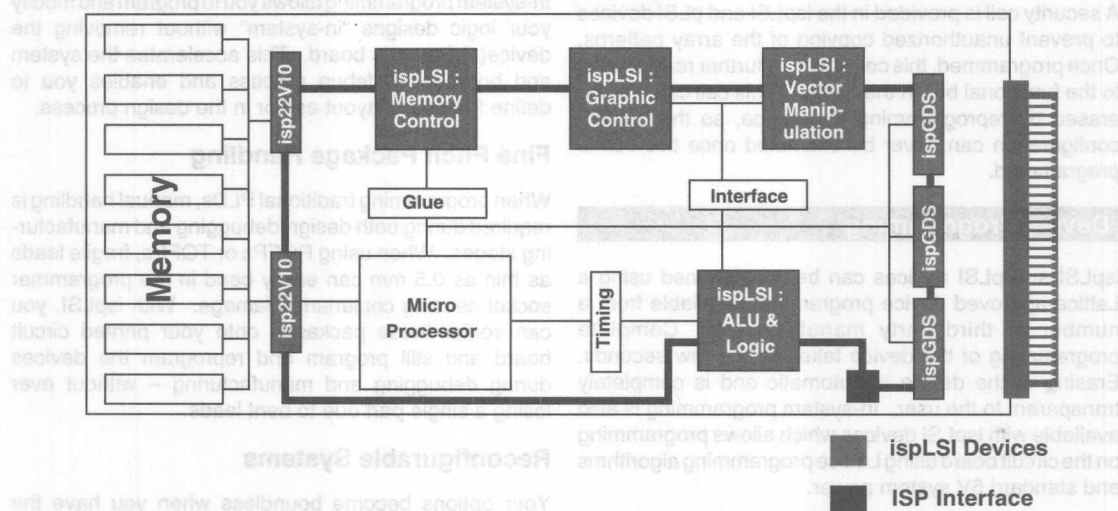
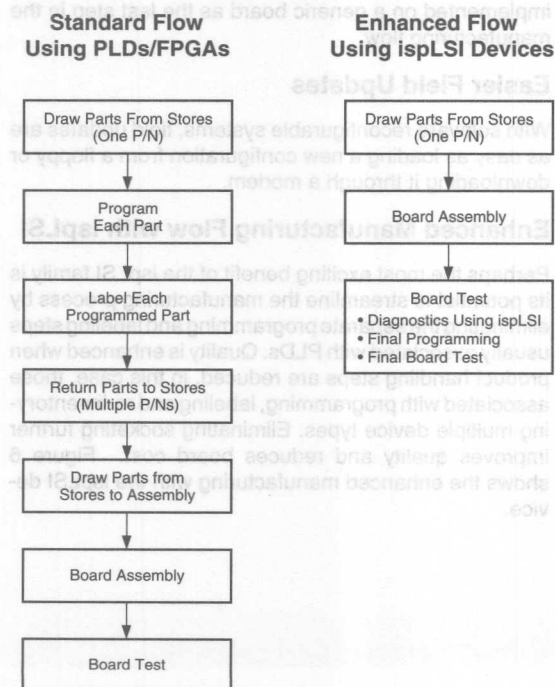
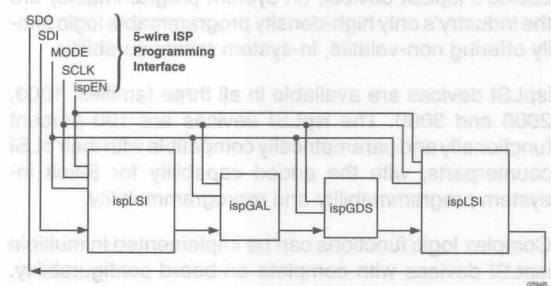


Figure 6. Manufacturing Flow Comparison



All necessary programming is achieved via five TTL-level logic interface signals (see figure 7). These five signals control the on-chip programming circuitry, which protects against inadvertent reprogramming via on-chip state machines. The ispLSI family can also be programmed using popular third-party logic programmers.

Figure 7. In-System Programming Interface (Multi-Chip Solution)



Boundary Scan

An emerging trend in board-level testing is boundary scan test, an attractive feature helping designers test system boards efficiently while lowering test and manufacturing costs. The ispLSI and pLSI 3000 family offers dedicated IEEE 1149.1 boundary scan support for all test functions required by the standard. By using ispLSI and pLSI devices you not only eliminate expensive "bed-of-nails" testers but also simplify testing of surface-mount boards, multi-layer boards and boards using fine-pitch packages. Boundary scan is ideal wherever tight board layout limits access to logic signals.

It only takes 4 pins to implement the boundary scan interface. The ispLSI 3000 devices share the four boundary scan signals with the in-system programming pins. This enhances the testability of system designs allowing logic to be reconfigured to improve controllability and observability.

Lattice Development Systems

The Lattice pLSI and ispLSI Development System (pDS) software is used to implement designs in ispLSI and pLSI devices. Design alternatives can be quickly implemented using Lattice's low cost pDS Software or the pDS+ family of Fitters that interface with third-party development software packages. This section describes the pDS and pDS+ Development Systems. Programmer support is also discussed.

pLSI and ispLSI Development System (pDS)

Features

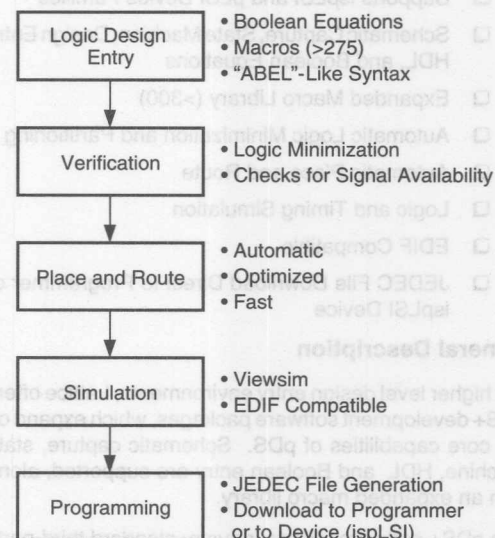
- ☐ High-Performance, Low-Cost Development Environment
- ☐ Supports ispLSI and pLSI Device Families
- ☐ Boolean Logic and Text File Design Entry
- ☐ Windows Based Graphical User Interface
- ☐ Over 275 Macros Available
- ☐ Automatic Place and Route
- ☐ Static Timing Table
- ☐ Logic Simulation with Viewlogic Viewsim
- ☐ JEDEC File Download Direct to Programmer or ispLSI Device

General Description

All ispLSI and pLSI families are supported by Lattice's low-cost pDS Software. It runs on IBM-compatible (386/486/Pentium) PCs with Microsoft Windows.

The graphical user interface employs an easy-to-use mouse and pull-down menu driven approach. Combined with Boolean logic data entry using an ABEL-like syntax, pDS makes design entry with ispLSI and pLSI quick and straightforward (see figure 8).

Figure 8. pDS Design Flow



The pDS Software supports over 275 macros to assist the design process. These macros cover most TTL functions, from gate primitives to 16-bit counters. The software also supports user-definable macros which can be modifications of existing macros or custom creations.

The pDS Software automatically verifies the design, performs logic minimization and checks for signal availability.

The Lattice Place and Route software assigns pins and critical speed paths while routing the design.

Quick compilation speeds the design, debug and rework process dramatically. Incremental design techniques are also supported.

Introduction to ispLSI and pLSI

Timing and functional simulation is available from Lattice, using Viewsim simulation software.

The Windows graphical user interface makes programming easy, using pull-down menus, intuitive point-and-click commands and self explanatory instructions. Without any up-front training, designs can be completed within hours instead of days or weeks.

pLSI and ispLSI Development System Plus (pDS+)

Features

- ☐ Supports ispLSI and pLSI Device Families
- ☐ Schematic Capture, State Machine, Design Entry HDL, and Boolean Equations
- ☐ Expanded Macro Library (>300)
- ☐ Automatic Logic Minimization and Partitioning
- ☐ Automatic Place and Route
- ☐ Logic and Timing Simulation
- ☐ EDIF Compatible
- ☐ JEDEC File Download Direct to Programmer or ispLSI Device

General Description

For higher level design entry environments, Lattice offers pDS+ development software packages, which expand on the core capabilities of pDS. Schematic capture, state machine, HDL and Boolean entry are supported, along with an expanded macro library.

The pDS+ software utilizes industry standard third-party design environments such as Viewlogic's Viewdraw and Data I/O's ABEL.

Running on IBM compatible (386/486/Pentium) PCs or workstation platforms, pDS+ software supports automatic logic minimization and partitioning as well as place and route, resulting in high logic utilization.

For logic and timing simulation, support is available from Lattice through Viewlogic Viewsim simulation tools.

Third Party Programming Support

The ispLSI and pLSI families are supported by popular third-party logic programmers including Data I/O, Logical Devices, BP-Microsystems, Stag, System General, SMS Micro Systems and Advin. Table 2 describes each vendor's specific programmer models that support the ispLSI and

pLSI devices. No proprietary, expensive, high pin-count programmers are required.

High pin-count socket adapters are available from Emulation Technology, Procon Technology, EDI Corporation and Logical Systems Corporation.

Additionally, the ispLSI family can be programmed on the board (in-system), which eliminates the need for a stand-alone programmer. For specific details refer to the Lattice Programming Tools Guide available from your local Sales Representative.

Table 2. Programming Support

Programmer Vendor	Model
Advin Systems	Pilot-U84
	Pilot-U40
	Pilot-GL/GCE
BP Microsystems	PLD-1128
	CP-1128
Data I/O	2900
	3900
	Unisite 40/48
Logical Devices	Allpro 40
	Allpro 88
SMS Micro Systems	Sprint Expert
Stag	System 3000
	ZL30/A
System General	TURPRO-1

isp Engineering Kit

The ispLSI family may also be programmed with Lattice's isp Engineering Kit Model 100 for PCs and Model 200 for Sun workstations. The kit is designed for engineering purposes only and is not intended for production use. By connecting an 8 wire cable to the parallel printer port of a PC, JEDEC files can be easily downloaded into the ispLSI device. Additionally, this cable can be connected directly to the circuit board facilitating on-board in-system programming.

1000 Family Architectural Description

ispLSI and pLSI 1000 Family Introduction

The basic unit of logic for the ispLSI and pLSI families is the Generic Logic Block (GLB). Figure 1 illustrates the pLSI 1032 with its 32 GLBs labelled A0, A1 .. D7. Each GLB has 18 inputs, a programmable AND/OR/XOR array, and four outputs which can be configured to be either combinatorial or registered. Inputs to the GLB come from the Global Routing Pool (GRP) and dedicated inputs. All of the GLB outputs are brought back into the GRP so that they can be connected to the inputs of any other GLB on the device.

As an example, the pLSI 1032 has 64 I/O cells, each of which is directly connected to an I/O pin. Each I/O cell can be individually programmed to be a combinatorial input, registered input, latched input, output or bi-directional I/O pin with 3-state control. Additionally, all outputs are polarity selectable, active high or active low. The signal levels are TTL compatible voltages and the output drivers can source 4 mA or sink 8 mA.

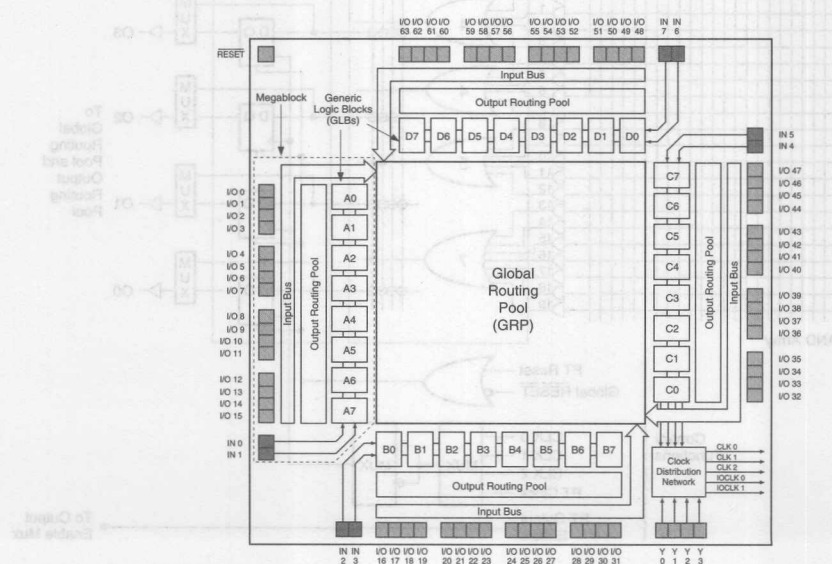
The I/O cells are grouped into sets of 16 as shown in figure 1. Each of these I/O groups is associated with a Megablock through the use of the Output Routing Pool (ORP).

Eight GLBs, 16 I/O cells, one ORP and two dedicated inputs are connected together to make a Megablock. The outputs of the eight GLBs are connected to a set of 16 universal I/O cells by the ORP. Each megablock shares a common Output Enable (OE) signal. The pLSI 1032 device, shown in figure 1, contains four Megablocks.

The GRP has as its inputs the outputs from all of the GLBs and all of the inputs from the bi-directional I/O cells. All of these signals are made available to the inputs of the GLBs. Delays through the GRP have been equalized to minimize timing skew.

Clocks in the devices are selected using the Clock Distribution Network. The dedicated clock pins (Y0, Y1, Y2 and Y3) are brought into the distribution network, and five outputs (CLK 0, CLK1, CLK 2, IOCLK 0 and IOCLK 1) are provided to route clocks to the GLBs and I/O cells. The Clock Distribution Network can also be driven from a special GLB (C0 on the ispLSI and pLSI 1032 device). The logic of this GLB allows the user to create an internal clock from a combination of internal signals within the device.

Figure 1. pLSI 1032 Functional Block Diagram



1000 Family Architectural Description

Generic Logic Block

The Generic Logic Block (GLB) is the standard logic block of the Lattice high-density ispLSI and pLSI devices. A GLB has 18 inputs, four outputs and the logic necessary to implement most standard logic functions. The internal logic of the GLB is divided into four separate sections: the AND Array, the Product Term Sharing Array (PTSA), the Reconfigurable Registers, and the Control Functions (see figure 2). The AND array consists of 20 product terms which can produce the logical sum of any of the 18 GLB inputs. Sixteen of the inputs come from the Global Routing Pool, and are either feedback signals from any of the GLBs or inputs from the external I/O cells. The two remaining inputs come directly from two dedicated input pins. These signals are available to the product terms in both the logical true and the complemented forms which makes boolean logic reduction more efficient.

The PTSA takes the 20 product terms and routes them to the four GLB outputs. There are four OR gates, with four, four, five and seven product terms each (see figure

2). The output of any of these OR gates can be routed to any of the four GLB outputs, and if more product terms are needed, the PTSA can combine them as necessary. In addition, the PTSA can share product terms similar to an FPLA device. If the user's main concern is speed, the PTSA can use a bypass circuit which provides four product terms to each output, to increase the performance of the cell (see figure 3). This can be done to any or all of the four outputs from the GLB.

The Reconfigurable Registers consist of four D-type flip-flops with an XOR gate on the input. The XOR gate in the GLB can be used either as a logic element or to reconfigure the D-type flip-flop to emulate a J-K or T-type flip-flop (see figure 4). This greatly simplifies the design of counters, comparators and ALU type functions. The registers can be bypassed if the user needs a combinatorial output. Each register output is brought back into the Global Routing Pool and is also brought to the I/O cells via the Output Routing Pool. Reconfigurable registers are not available when the four product term bypass is used.

Figure 2. GLB: Product Term Sharing Array Example

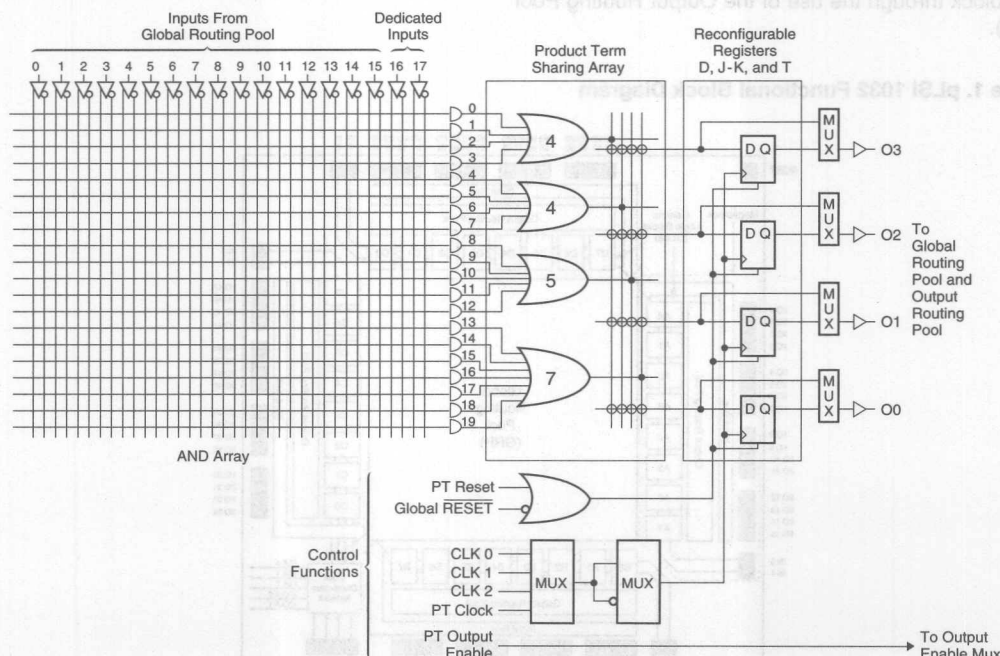


Figure 3. GLB: Four Product Term Bypass Example

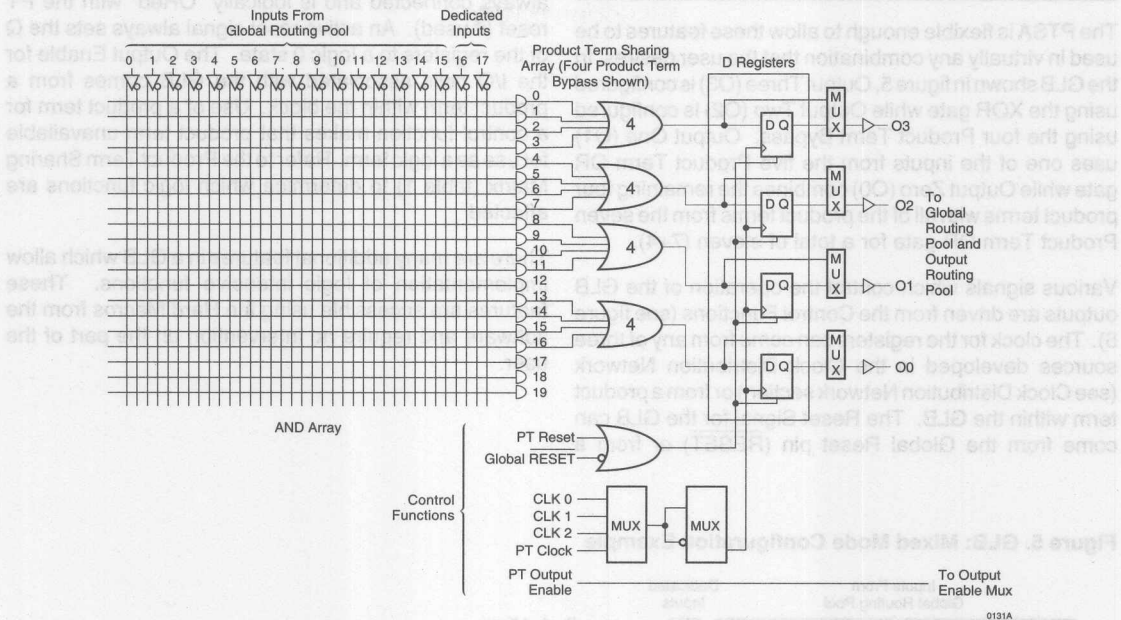
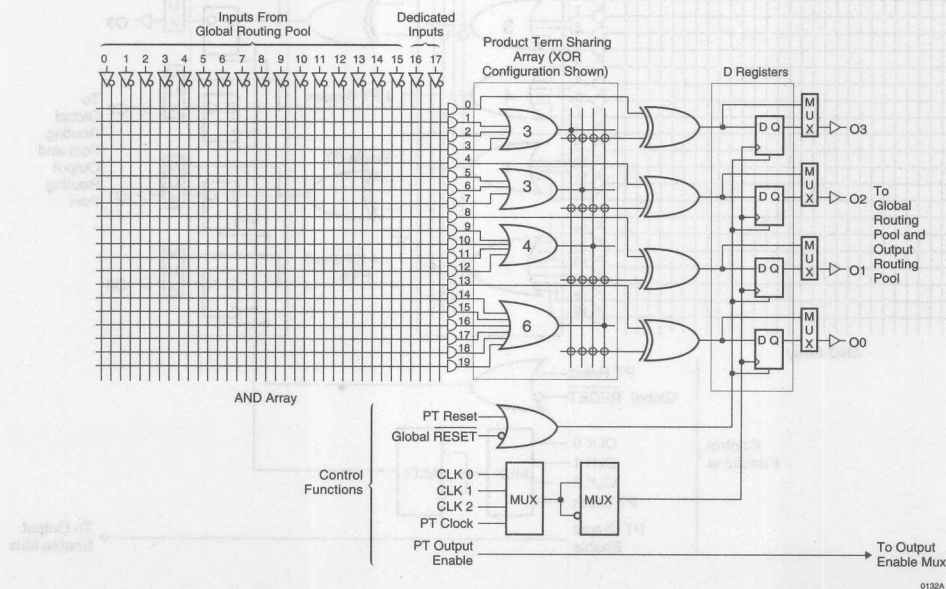


Figure 4. GLB: XOR Gate Example



1000 Family Architectural Description

Generic Logic Block (continued)

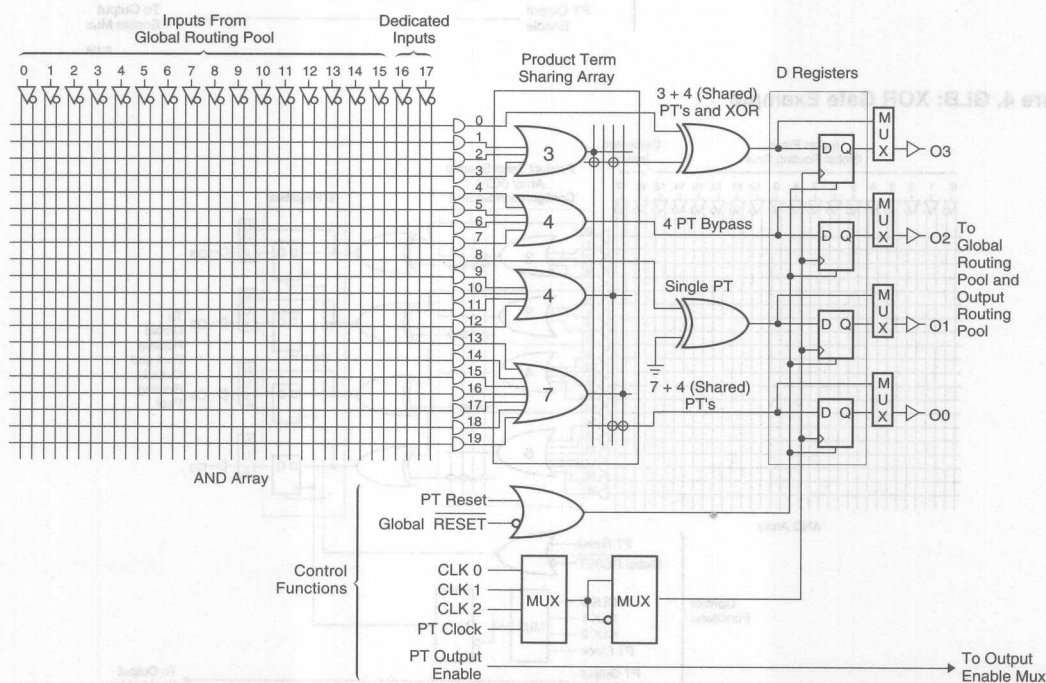
The PTSA is flexible enough to allow these features to be used in virtually any combination that the user desires. In the GLB shown in figure 5, Output Three (O3) is configured using the XOR gate while Output Two (O2) is configured using the four Product Term Bypass. Output One (O1) uses one of the inputs from the five Product Term OR gate while Output Zero (O0) combines the remaining four product terms with all of the product terms from the seven Product Term OR gate for a total of eleven (7+4).

Various signals which control the operation of the GLB outputs are driven from the Control Functions (see figure 5). The clock for the registers can come from any of three sources developed in the Clock Distribution Network (see Clock Distribution Network section) or from a product term within the GLB. The Reset Signal for the GLB can come from the Global Reset pin (RESET) or from a

product term within the block. The global reset pin is always connected and is logically "ORed" with the PT reset (if used). An active reset signal always sets the Q of the registers to a logic 0 state. The Output Enable for the I/O cells associated with the GLB comes from a product term within the block. Use of a product term for a control function makes that product term unavailable for use as a logic term. Refer to the Product Term Sharing Matrix (table 1) to determine which logic functions are affected.

There are many additional features in a GLB which allow implementation of logic intensive functions. These features are accessible using the Hard Macros from the software and require no intervention on the part of the user.

Figure 5. GLB: Mixed Mode Configuration Example



1000 Family Architectural Description

Product Term Sharing Matrix

This matrix shows how each of the product terms are used in the various modes. As an example, Product Term 12 can be used as an input to the five input OR gate in the standard configuration. This OR gate under standard configuration can be routed to any of the four GLB outputs. Product Term 12 is not used in the four product

term bypass mode. When GLB output one is used in the XOR mode Product Term 12 becomes one of the inputs to the four input OR Gate. If Product Term 12 is not used in the logic, then it is available for use as either the Asynchronous Clock signal or the GLB Reset signal.

2

Table 1. Product Term Sharing Matrix

Product Term #	Standard Configuration Output Number	Four Product Term Bypass Output Number	Single Product Term Output Number	XOR Function Output Number	Alternate Function
	3 2 1 0	3 2 1 0	3 2 1 0	3 3 2 2 1 1 0 0	
0	■	■	■	■	
1	■	■	■	■	
2	■	■	■	■	
3	■	■	■	■	
4	■	■	■	■	
5	■	■	■	■	
6	■	■	■	■	
7	■	■	■	■	
8	■	■	■	■	
9	■	■	■	■	
10	■	■	■	■	
11	■	■	■	■	
12	■	■	■	■	■ CLK/Reset
13	■	■	■	■	
14	■	■	■	■	
15	■	■	■	■	
16	■	■	■	■	
17	■	■	■	■	
18	■	■	■	■	
19	■	■	■	■	■ OE/Reset

The Megablock

A Megablock consists of eight GLBs, an ORP, 16 I/O cells, two dedicated inputs and a common product term OE. Each of these will be explained in detail in the following sections. These elements are coupled together as shown in figure 6. The various members of the ispLSI and pLSI families combine from one to eight Megablocks on a single device (see table 2).

For the 1000 Family, the eight GLBs within the Megablock share two dedicated input pins. These dedicated input pins are not available to GLBs in any other Megablock. These pins are dedicated (non-registered) inputs only

and are automatically assigned by software. The product term OE signal is generated within the Megablock and is common to all 16 of the I/O cells in the Megablock. The OE signal can be generated using a product term (PT19) in any of the eight GLBs within the Megablock (see the section on the Output Enable Control for further details).

Because of the shared logic within the Megablock, signals which share a common function (counters, busses, etc.) should be grouped within a Megablock. This will allow the user to obtain the best utilization of the logic within the device and eliminate routing bottlenecks.

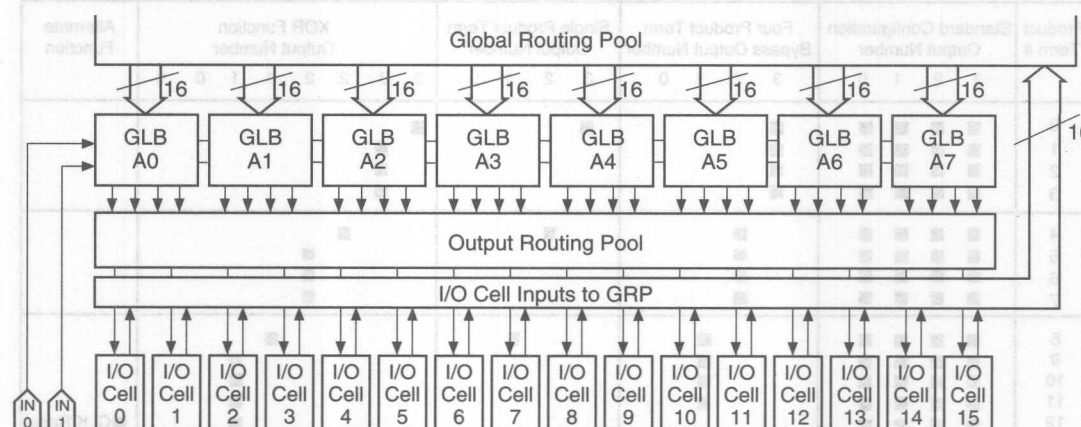
1000 Family Architectural Description

Table 2. Device Resources

ispLSI and pLSI Devices	Megablocks	GLBs	I/O Cells	Dedicated Inputs
1016	2	16	32	4
1024	3	24	48	6
1032	4	32	64	8
1048/1048C	6	48	96	10/12

Table 2-0015B

Figure 6. The Megablock Block Diagram



Input Routing

Signal inputs are handled in two ways within the device. First, each I/O cell within the device has its input routed directly to the GRP. This gives every GLB within the device access to each I/O cell input. Second, each Megablock has two dedicated inputs which are directly routed to the eight GLBs within the Megablock. Both input paths are shown in figure 6.

The Output Routing Pool

The ORP routes signals from the GLB outputs to I/O cells configured as outputs or bi-directional pins (see figure 7). The purpose of the ORP is to allow greater flexibility when assigning I/O pins. It also simplifies the job for the routing software which results in a higher degree of utilization.

By examining the ORP in figure 7, it can be seen that a GLB output can be connected to one of four I/O cells. Further flexibility is provided by using the PTSA, (figures 2 through 5) which makes the GLB outputs completely interchangeable. This allows the routing program to freely interchange the outputs to achieve the best routability. This is an automatic process and requires no intervention on the part of the user.

The ORP bypass connections (see figure 8) further increase the flexibility of the device. The ORP bypass connects specific GLB outputs to specific I/O cells at a faster speed. The bypass path tends to restrict the routability of the device and should only be used for critical signals.

Figure 7. Output Routing Pool

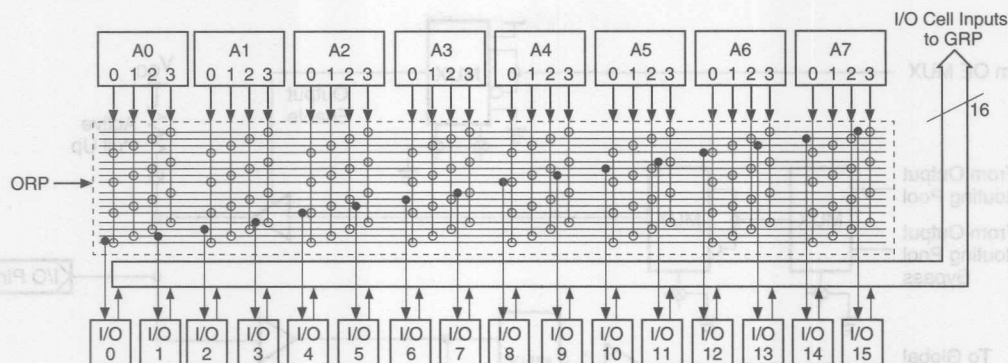
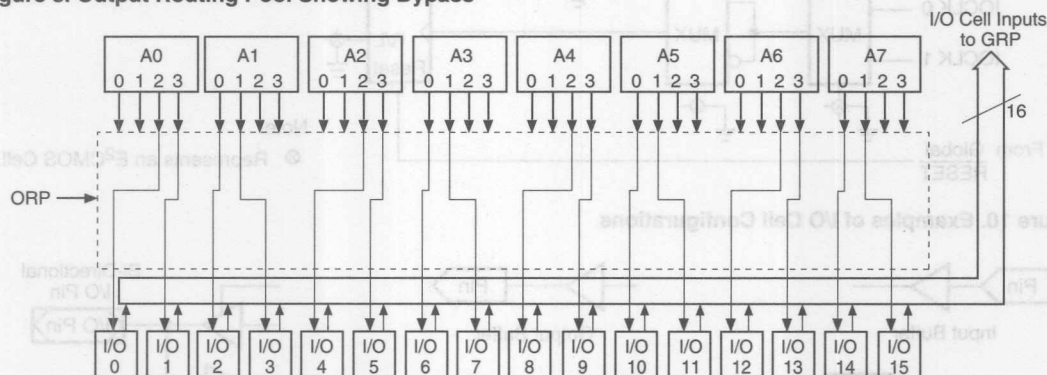


Figure 8. Output Routing Pool Showing Bypass



I/O Cell

The I/O cell (see figure 9) is used to route input, output or bi-directional signals connected to the I/O pin. The two logic inputs come from the ORP (see figure 9). One comes from the ORP, and the other comes from the faster ORP bypass. A pair of multiplexers select which signal will be used, and its polarity. The Output Enable of the I/O cell is controlled by the OE signal generated within each Megablock.

As with the data path, a multiplexer selects the signal polarity. The Output Enable can be set to a logic high (enabled) when an output pin is desired, or logic low (disabled) when an input pin is needed. The Global Reset (RESET) signal is driven by the active low chip reset pin. This reset is always connected to all GLB and I/O registers. Each I/O cell can individually select one of

the two clock signals (IOCLK 0 or IOCLK 1). These clock signals are generated by the Clock Distribution Network.

Using the multiplexers, the I/O cell can be configured as an input, an output, a 3-stated output or a bi-directional I/O. The D-type register can be configured as a level sensitive transparent latch or an edge triggered flip-flop to store the incoming data. Figure 10 illustrates some of the various I/O cell configurations possible.

There is an active pull-up resistor on the I/O pins which is automatically used when the pin is not connected. An option exists to have active pull-up resistors connected to all pins. This improves the noise immunity and reduces Icc for the device.

1000 Family Architectural Description

Figure 9. I/O Cell Architecture

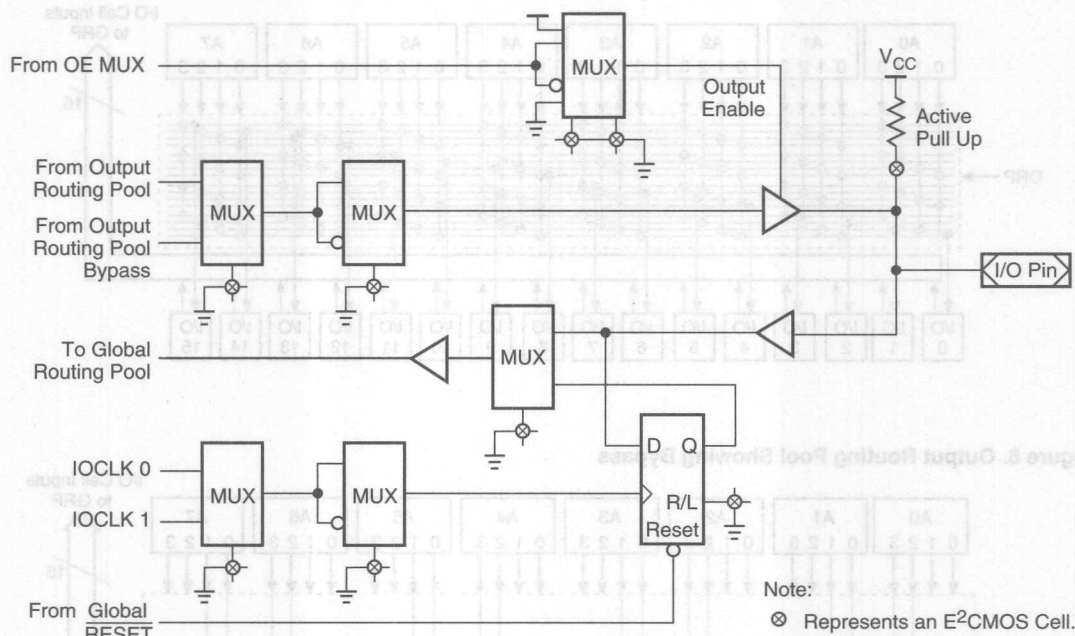
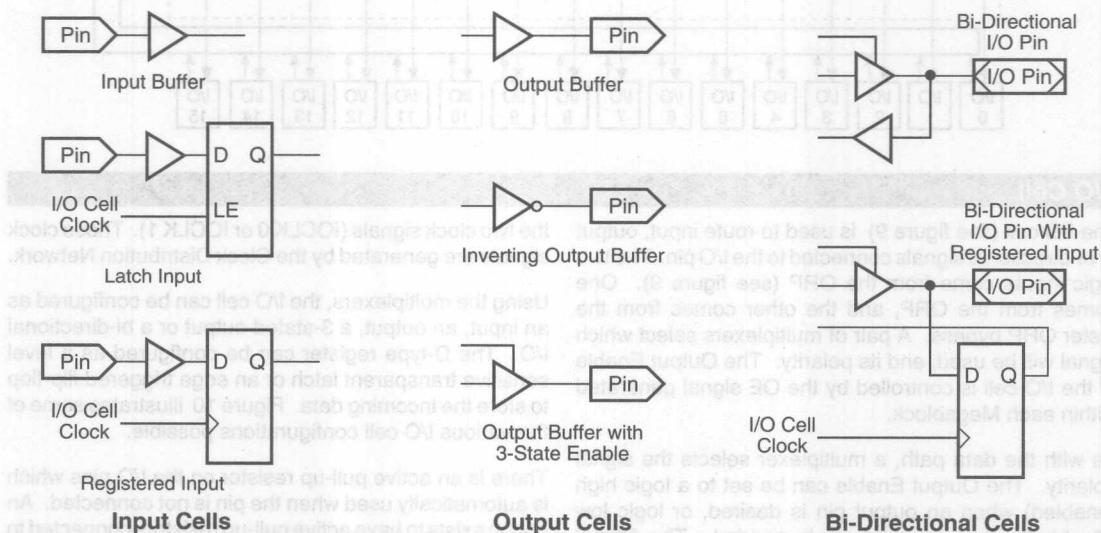


Figure 10. Examples of I/O Cell Configurations



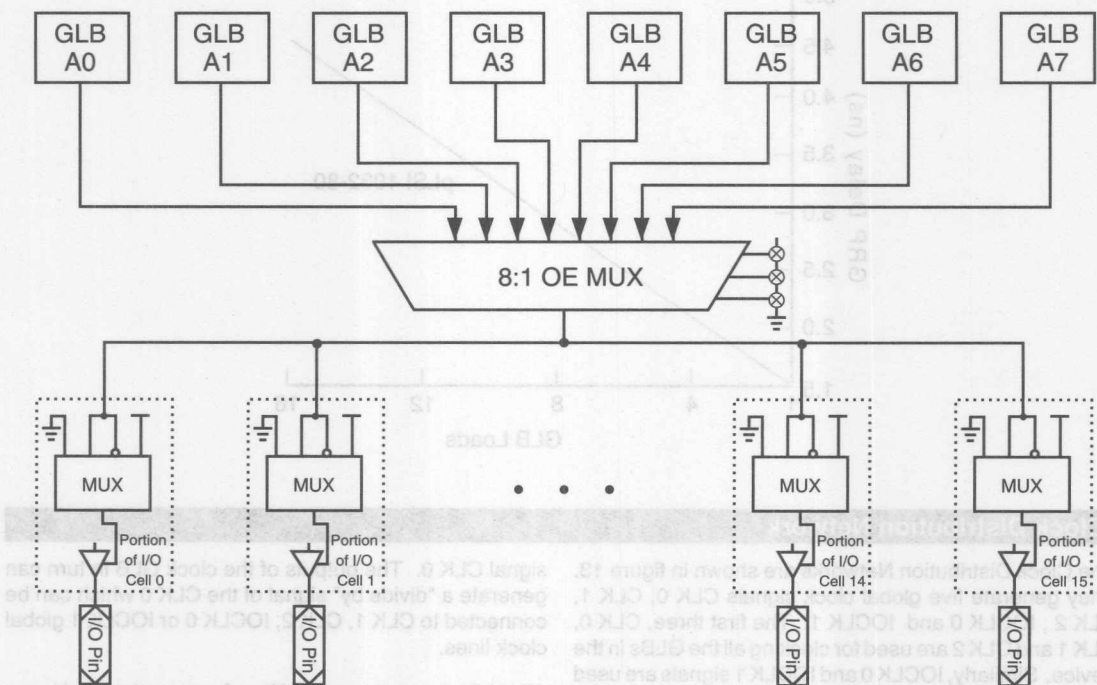
The Output Enable Control

One OE signal can be generated within each GLB using the OE Product Term (PT19). One of the eight OE signals within a Megablock is then routed to all of the I/O cells within that Megablock (see figure 11). This OE signal can simultaneously control all of the 16 I/O cells which are used in 3-state mode. Individual I/O cells also have independent control for permanently enabling or

disabling the output buffer (refer to the I/O cell section). Only one OE signal is allowed per Megablock for 3-state operation. The advantage to this approach is that the OE signal can be generated in any GLB within the Megablock which happens to have an unused OE product term. This frees up the other OE product terms for use as logic.

2

Figure 11. Output Enable Control for a Megablock



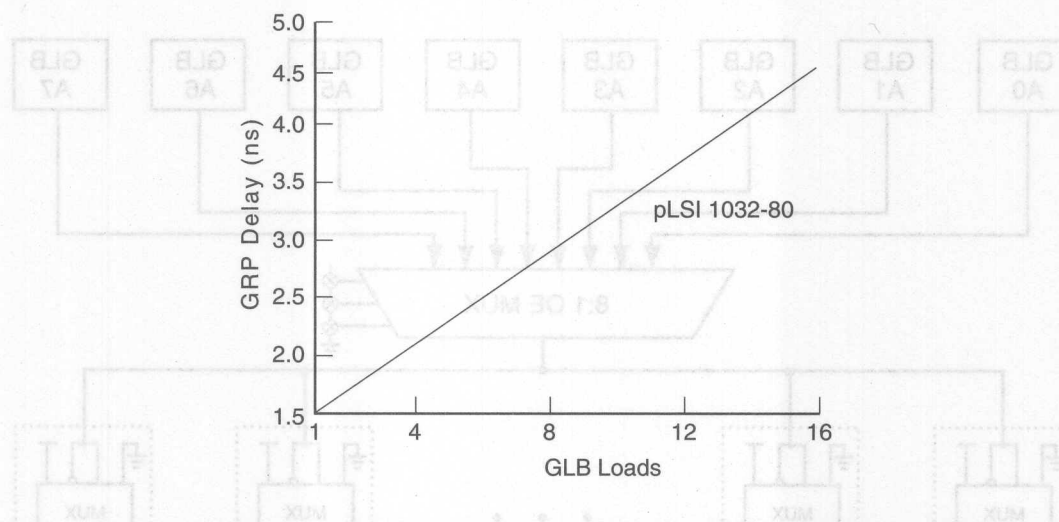
1000 Family Architectural Description

Global Routing Pool

The GRP is a Lattice proprietary interconnect structure which offers fast predictable speeds with complete connectivity. The GRP allows the outputs from the GLBs or the I/O cell inputs to be connected to the inputs of the GLBs. Any GLB output is available to the input of all other GLBs, and similarly an input from an I/O pin is available

as an input to all of the GLBs. Because of the uniform architecture of the ispLSI and pLSI devices, the delays through the GRP are both consistent and predictable. However, they are slightly affected by GLB loading as shown in the example pLSI 1032-80 GLB Loading Delay graph (see figure 12).

Figure 12. Example Graph of GRP Delay vs GLB Loading



Clock Distribution Network

The Clock Distribution Networks are shown in figure 13. They generate five global clock signals CLK 0, CLK 1, CLK 2, IOCLK 0 and IOCLK 1. The first three, CLK 0, CLK 1 and CLK 2 are used for clocking all the GLBs in the device. Similarly, IOCLK 0 and IOCLK 1 signals are used for clocking all of the I/O cells in the device. There are four dedicated system clock pins (Y0, Y1, Y2, Y3), three for the ispLSI and pLSI 1016 (Y0, Y1, Y2), which can be directed to any GLB or any I/O cell using the Clock Distribution Network. The other inputs to the Clock Distribution Network are the four outputs of a dedicated clock GLB ("C0" for the pLSI 1032 is shown in figure 1). These clock GLB outputs can be used to create a user-defined internal clocking scheme.

For example, the clock GLB can be clocked using the external main clock pin Y0 connected to global clock

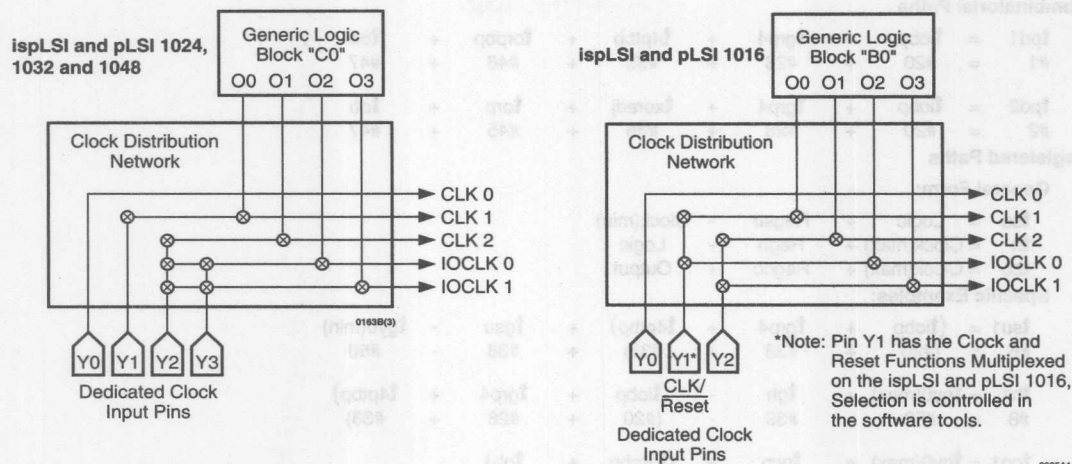
signal CLK 0. The outputs of the clock GLB in turn can generate a "divide by" signal of the CLK 0 which can be connected to CLK 1, CLK 2, IOCLK 0 or IOCLK 1 global clock lines.

All GLBs have the capability of generating their own asynchronous clocks using the clock Product Term (PT12). CLK 0, CLK 1 and CLK 2 feed to their corresponding clock MUX inputs on all the GLBs (see figure 2).

The two I/O clocks generated in the Clock Distribution Network IOCLK 0 and IOCLK 1, are brought to all the I/O cells and the user programs the I/O cell to use one of the two.

1000 Family Architectural Description

Figure 13. Clock Distribution Networks

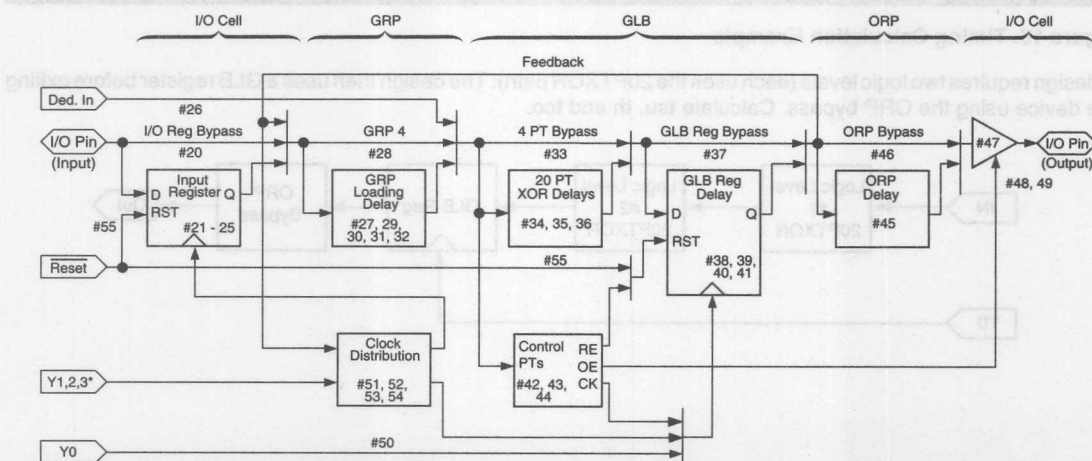


Timing Model

The task of determining the timing through the device is simple and straightforward. A device timing model is shown in figure 14. To determine the time that it takes for data to propagate through the device, simply determine the path the data is expected to follow, and add the various delays together (figure 15). Critical timing paths

are shown in figure 14, using data sheet parameters. Note that the Internal timing parameters are given for reference only, and are not tested. (External timing parameters are tested and guaranteed on every device).

Figure 14. ispLSI and pLSI Timing Model¹



*Note: Y1 and Y2 only for the ispLSI and pLSI 1016.

1000 Family Architectural Description

Figure 15. ispLSI and pLSI Timing Model Examples¹

Combinatorial Paths

$$\begin{aligned} t_{pd1} &= t_{iobp} + t_{grp4} + t_{4ptbp} + t_{torbp} + t_{ob} \\ \#1 &= \#20 + \#28 + \#33 + \#46 + \#47 \\ t_{pd2} &= t_{iobp} + t_{grp4} + t_{xoradj} + t_{torp} + t_{ob} \\ \#2 &= \#20 + \#28 + \#36 + \#45 + \#47 \end{aligned}$$

Registered Paths

General Form:

$$\begin{aligned} t_{su} &= \text{Logic} + \text{Regsu} - \text{Clock(min)} \\ t_h &= \text{Clock(max)} + \text{Regh} - \text{Logic} \\ t_{co} &= \text{Clock(max)} + \text{Regco} + \text{Output} \end{aligned}$$

Specific Examples:

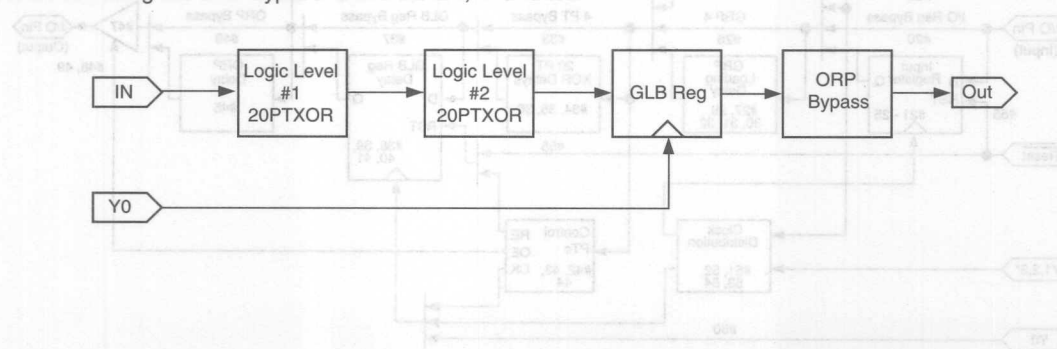
$$\begin{aligned} t_{su1} &= (t_{iobp} + t_{grp4} + t_{4ptbp}) + t_{gsu} - t_{gy0(min)} \\ \#6 &= (\#20 + \#28 + \#33) + \#38 - \#50 \\ t_{h1} &= t_{gy0(max)} + t_{gh} - (t_{iobp} + t_{grp4} + t_{4ptbp}) \\ \#8 &= \#50 + \#39 - (\#20 + \#28 + \#33) \\ t_{co1} &= t_{gy0(max)} + t_{gco} + (t_{torbp} + t_{ob}) \\ \#7 &= \#50 + \#40 + (\#46 + \#47) \\ t_{su2} &= (t_{iobp} + t_{grp4} + t_{xoradj}) + t_{gsu} + t_{gy0(min)} \\ \#9 &= (\#20 + \#28 + \#36) + \#38 + \#50 \\ t_{h2} &= t_{gy0(max)} + t_{gh} - (t_{iobp} + t_{grp4} + t_{xoradj}) \\ \#11 &= \#50 + \#39 - (\#20 + \#28 + \#36) \\ t_{co2} &= t_{gy0(max)} + t_{gco} + (t_{torp} + t_{ob}) \\ \#10 &= \#50 + \#40 + (\#45 + \#47) \end{aligned}$$

1. The timing parameter reference numbers refer to the Internal Timing Parameters contained in the individual data sheets.

Circuit Timing Example

Figure 16. Timing Calculation Example

A design requires two logic levels (each uses the 20PTXOR path). The design then uses a GLB register before exiting the device using the ORP bypass. Calculate t_{su} , t_h and t_{co} .



1000 Family Architectural Description

Figure 16. Timing Calculation Example (continued)

$$\begin{aligned}
 t_{su} &= \text{Logic} + \text{Reg su} - \text{Clock (min)} \\
 &= (t_{iobp} + t_{grp4} + t_{20ptxor} + t_{gbp} + t_{grp4} + t_{20ptxor}) + t_{gsu} - t_{gy0(\text{min})} \\
 &= (\#20 + \#28 + \#35 + \#37 + \#28 + \#35) + \#38 - \#50 \\
 19.5 \text{ ns} &= (2.0 + 2.0 + 8.0 + 1.0 + 2.0 + 8.0) + 1.0 - 4.5 \\
 \\
 t_h &= \text{Clock (max)} + \text{Reg h} - \text{Logic} \\
 &= t_{gy0(\text{max})} + t_{gh} - (t_{iobp} + t_{grp4} + t_{20ptxor} + t_{gbp} + t_{grp4} + t_{20ptxor}) \\
 &= \#50 + \#39 - (\#20 + \#28 + \#35 + \#37 + \#28 + \#35) \\
 -14.0 \text{ ns} &= 4.5 + 4.5 - (2.0 + 2.0 + 8.0 + 1.0 + 2.0 + 8.0) \\
 \\
 t_{co} &= \text{Clock (max)} + \text{Reg co} + \text{Output} \\
 &= t_{gy0(\text{max})} + t_{gco} + (t_{orpbp} + t_{ob}) \\
 &= \#50 + \#40 + (\#46 + \#47) \\
 10.0 \text{ ns} &= 4.5 + 2.0 + (0.5 + 3.0)
 \end{aligned}$$

1. The delay values used are for a pLSI 1032-80 device.

Notes

Figure 16. Timing Calculation Example (continued)

$$\begin{aligned}
 t_{\text{su}} &= \text{Logic} + \text{Flag su} - \text{Clock (min)} \\
 &= (\text{t}_{\text{logic}} + \text{t}_{\text{flag}} + \text{t}_{\text{clock}} + \text{t}_{\text{flag}} + \text{t}_{\text{clock}}) + \text{t}_{\text{flag}} - \text{t}_{\text{clock}} \\
 &= (250 + 450 + 450 + 450 + 450) + 450 - 450 \\
 &= (250 + 250 + 450 + 150 + 250 + 80) + 150 - 450 \\
 &= 1050 \text{ ns} \\
 t_{\text{tr}} &= \text{Clock (max)} + \text{Flag tr} - \text{Logic} \\
 &= \text{t}_{\text{clock}}(\text{max}) + \text{t}_{\text{flag}} - (\text{t}_{\text{logic}} + \text{t}_{\text{clock}} + \text{t}_{\text{flag}} + \text{t}_{\text{clock}} + \text{t}_{\text{flag}} + \text{t}_{\text{clock}}) \\
 &= 250 + 450 - (250 + 450 + 450 + 450 + 450 + 450) \\
 &= 450 + 450 - (250 + 250 + 450 + 150 + 250 + 80) \\
 &= -140 \text{ ns} \\
 t_{\text{co}} &= \text{Clock (max)} + \text{Flag co} + \text{Output} \\
 &= \text{t}_{\text{clock}}(\text{max}) + \text{t}_{\text{flag}} + (\text{t}_{\text{logic}} + \text{t}_{\text{flag}}) \\
 &= 250 + 450 + (450 + 450) \\
 &= 1600 \text{ ns}
 \end{aligned}$$

1. The delay values used are for a PLD 1032-80 device.

2000 Family Architectural Description

ispLSI and pLSI 2000 Family Introduction

The basic unit of logic of the ispLSI and pLSI 2000 family is essentially the same as that of the ispLSI and pLSI 1000 family. However, there are some specific architectural differences: Global clock structure, I/O Cell and OE structure, and ORP structure. A functional block diagram of the 2032 device is shown in figure 1. These architectural differences are described in detail below.

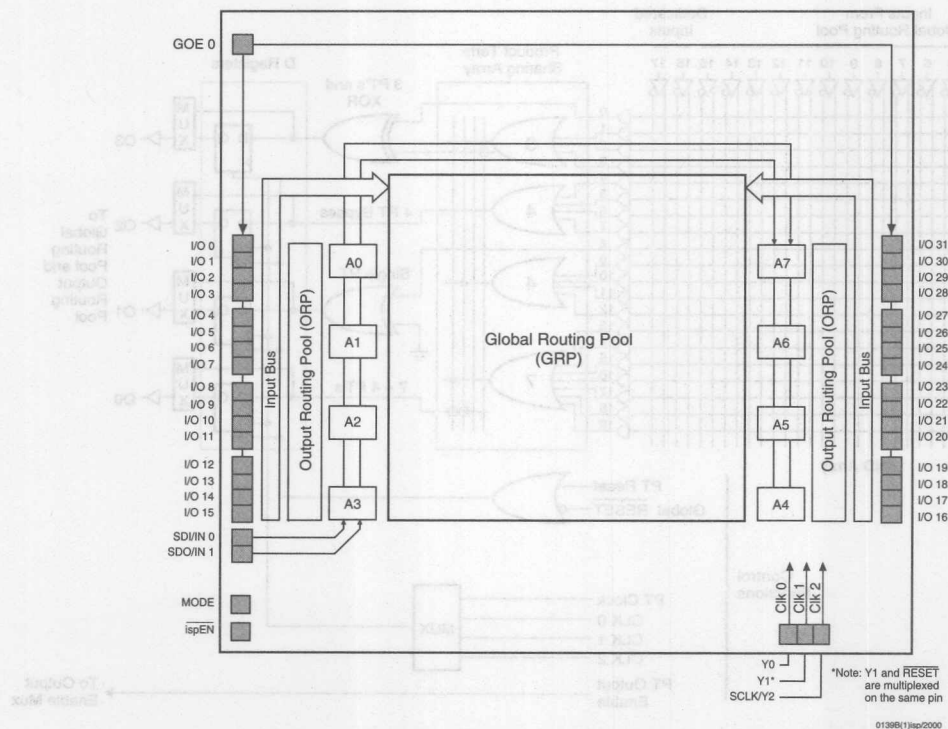
Global Clock Structure

The clock GLB distribution network of the 1000 family has been eliminated and replaced by three dedicated global

GLB clock input signals CLK0, CLK1, and CLK2. These three clocks are used for clocking all the GLBs configured as registers in the device. They feed directly to the GLB clock input via a clock multiplexer. CLK0 is associated with system clock pin Y0, CLK1 corresponds to system clock pin Y1, and finally CLK3 corresponds to system clock pin Y2. This is illustrated in figure 2. The GLB global clocks do not have inversion capability, but all GLBs continue to have the capability of generating their own asynchronous clocks using the clock product term (PT12) with inversion capability. The GLB global clocks and the GLB product term clock feed to their corresponding clock multiplexer shown in figure 3.

2

Figure 1. pLSI 2032 Functional Block Diagram



2000 Family Architectural Description

Figure 2. Global Clock Structure

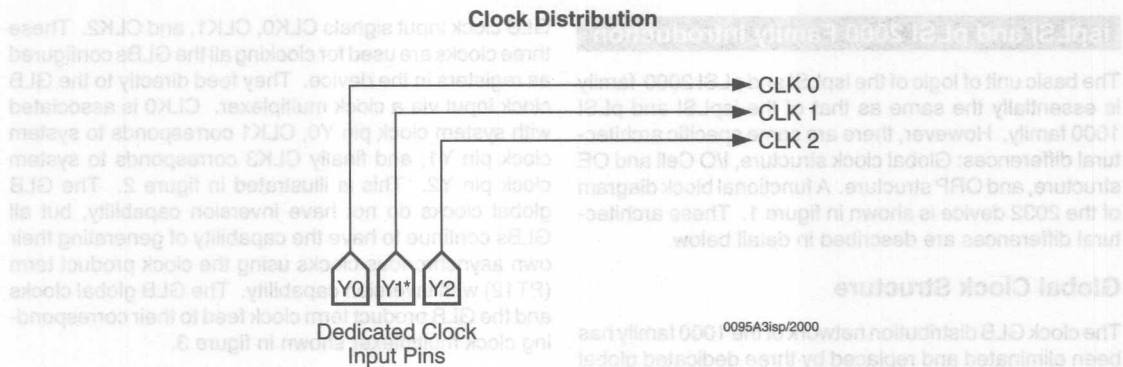
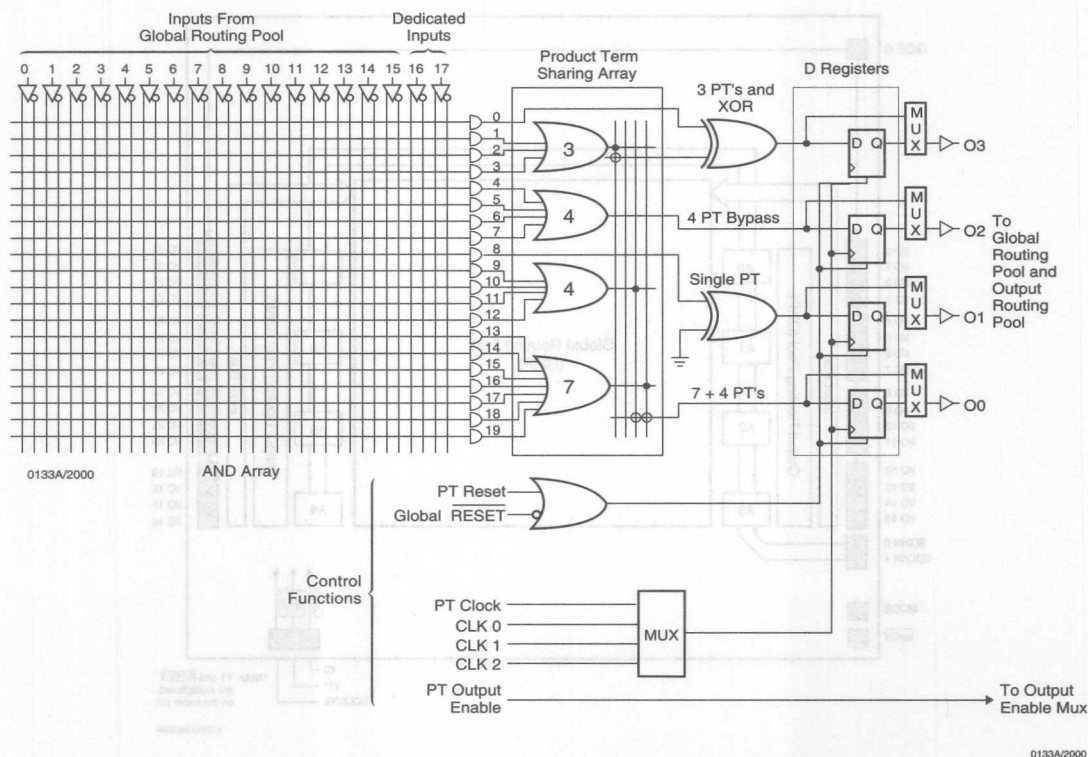


Figure 3. GLB with Clock Multiplexer Scheme



I/O Cell and OE Structure

The reconfigurable input register or latch has been removed to simplify the I/O cell architecture. Each I/O cell can be individually programmed to be a combinatorial input, combinatorial output, or a bi-directional I/O pin with 3-state control. With the simplified I/O cell architecture, the I/O clocks have also been removed. This is illustrated in figure 4. The product term output enable (PTOE)

signal is still generated within each GLB using product term 19. The PTOE is generated in one of the eight GLBs. In addition to the PTOE, there is a global output enable (GOE) pin which can control any of the device's 3-state output buffers. The multiplexing between the GOE and PTOE is illustrated in figure 5. The 2032 device has one GOE, and the 2064 and 2096 devices each have 2 GOEs.

Figure 4. ispLSI and pLSI 2000 Family I/O Cell Architecture

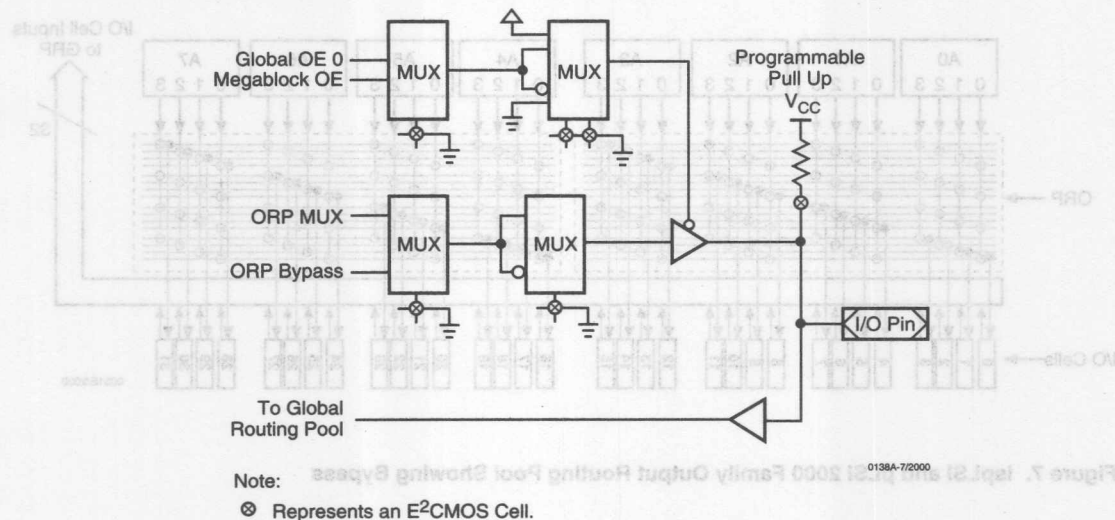
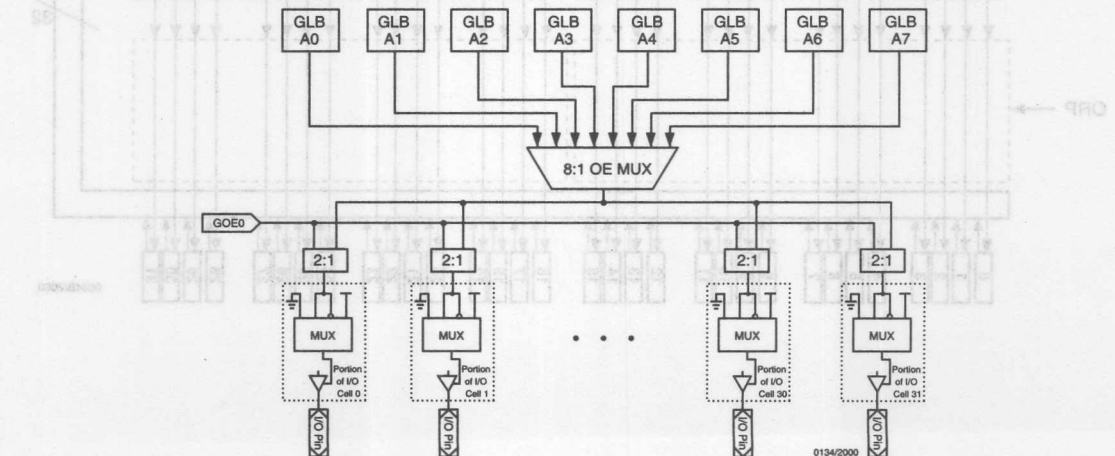


Figure 5. ispLSI and pLSI 2000 Family Output Enable Controls



2000 Family Architectural Description

Output Routing Pool (ORP)

Each megablock now contains two ORPs to increase output routability. A set of four GLBs is associated with one of the two ORPs within the megablock. The 16 outputs of the four GLBs within a megablock will feed to any of the 16 associated I/O cells. In the 1000 family, the

32 GLB outputs feed only 16 associated I/O cells. In this device family, 32 GLB outputs of a megablock can feed 32 I/O cells. Output routability has doubled. This is illustrated in figure 6. Each GLB output has an ORP bypass capability so more designs can have critical output signals. This is shown in figure 7.

Figure 6. ispLSI and pLSI 2000 Family Output Routing Pool

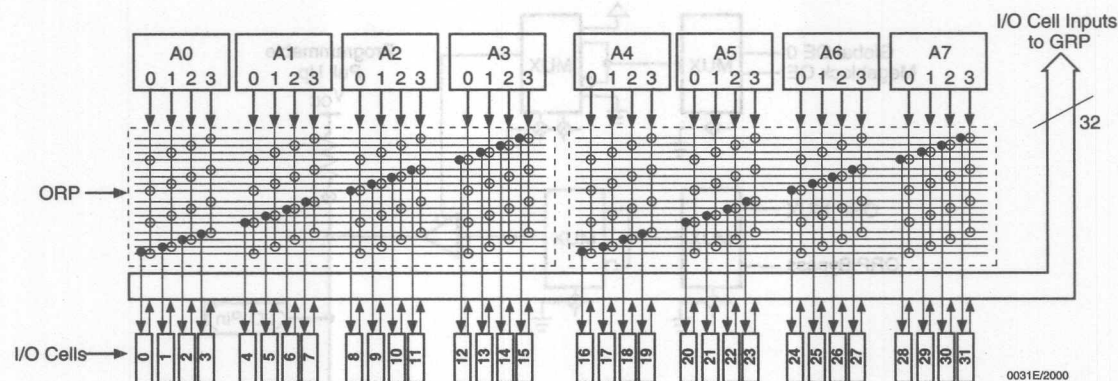
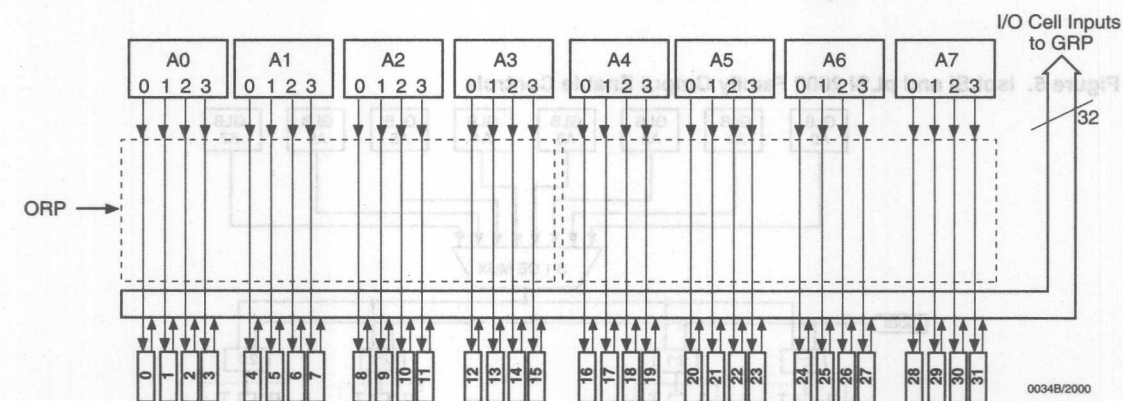


Figure 7. ispLSI and pLSI 2000 Family Output Routing Pool Showing Bypass



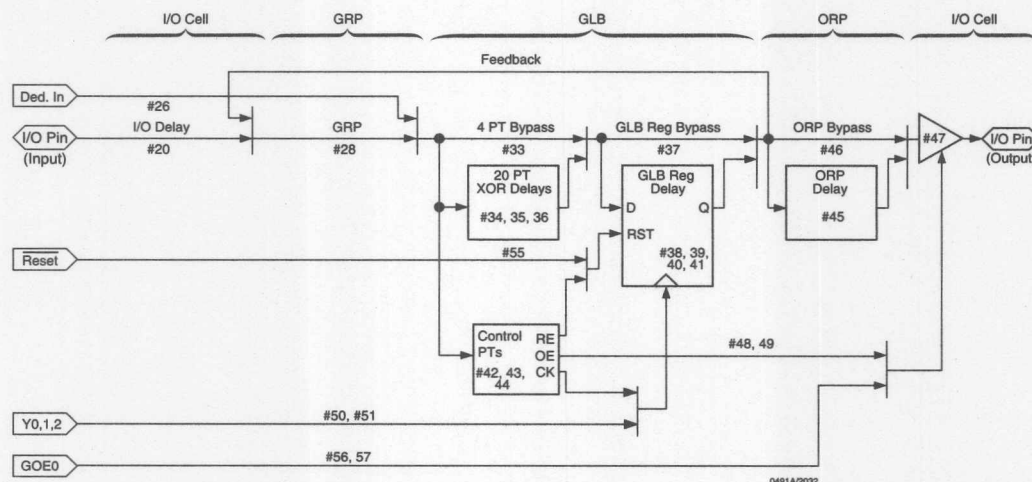
Timing Model

The task of determining the timing through the device is simple and straightforward. A device timing model is shown in figure 8. To determine the time that it takes for data to propagate through the device, simply determine the path the data is expected to follow, and add the various delays together (figure 8). Critical timing paths

are shown in figure 8, using data sheet parameters. Note that the Internal timing parameters are given for reference only, and are not tested. (External timing parameters are tested and guaranteed on every device).

2

Figure 8. ispLSI and pLSI 2032 Timing Model



Derivations of t_{su} , t_h and t_{co} from the Product Term Clock¹

$$\begin{aligned}
 t_{su} &= \text{Logic} + \text{Reg } su - \text{Clock (min)} \\
 &= (t_{iobp} + t_{grp} + t_{20ptxor}) + (t_{gsu}) - (t_{iobp} + t_{grp} + t_{ptck(min)}) \\
 &= (\#24 + \#28 + \#35) + (\#38) - (\#20 + \#28 + \#44) \\
 2.2 \text{ ns} &= (1.0 + 1.3 + 4.7) + (0.8) - (1.0 + 1.3 + 3.3) \\
 t_h &= \text{Clock (max)} + \text{Reg } h - \text{Logic} \\
 &= (t_{iobp} + t_{grp} + t_{ptck(max)}) + (t_{gh}) - (t_{iobp} + t_{grp} + t_{20ptxor}) \\
 &= (\#20 + \#28 + \#44) + (\#39) - (\#20 + \#28 + \#35) \\
 1.6 \text{ ns} &= (1.0 + 1.3 + 3.3) + (3.0) - (1.0 + 1.3 + 4.7) \\
 t_{co} &= \text{Clock (max)} + \text{Reg } co + \text{Output} \\
 &= (t_{iobp} + t_{grp} + t_{ptck(max)}) + (t_{gco}) + (t_{orp} + t_{ob}) \\
 &= (\#20 + \#28 + \#44) + (\#40) + (\#45 + \#47) \\
 8.8 \text{ ns} &= (1.0 + 1.3 + 3.3) + (0.7) + (1.3 + 1.2)
 \end{aligned}$$

1. Calculations are based upon timing specs for the ispLSI and pLSI 2032-135L

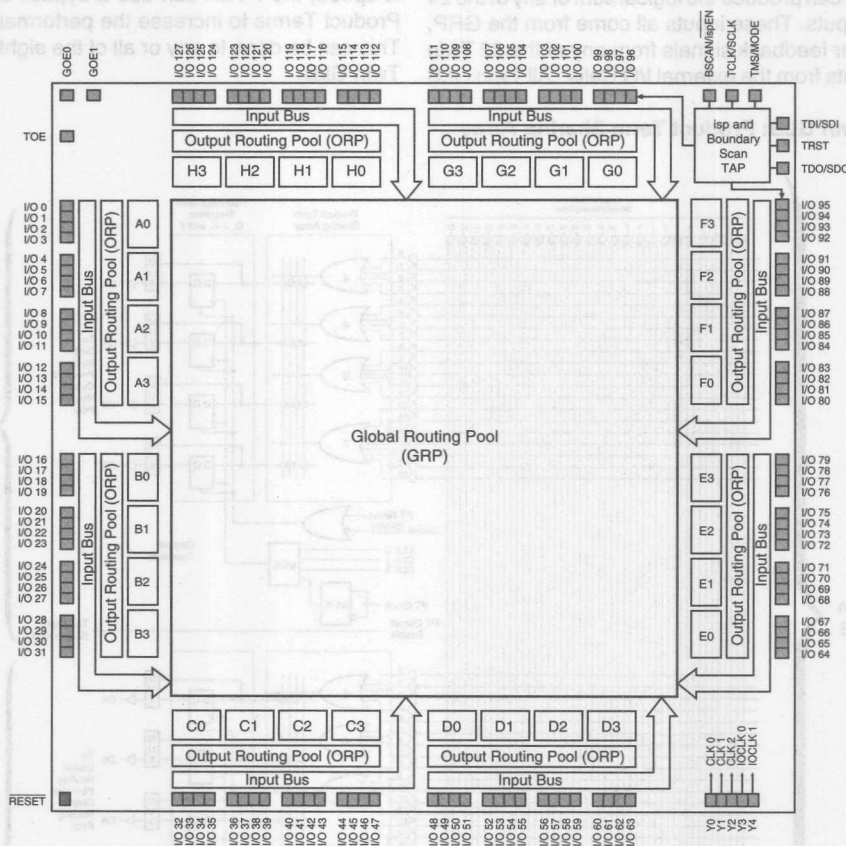
3000 Family Architectural Description

ispLSI and pLSI 3000 Family Introduction

The basic unit of logic of the ispLSI and pLSI 3000 family is closely related to that of the pLSI and ispLSI 1000 family. However, there are some notable architectural

differences: Boundary Scan, Megablock and GLB structure, Global clock structure, and I/O cell structure. A functional block diagram of the ispLSI 3256 device is shown in figure 1. The architectural differences are described in the following sections.

Figure 1. ispLSI 3256 Functional Block Diagram



3000 Family Architectural Description

Generic Logic Block

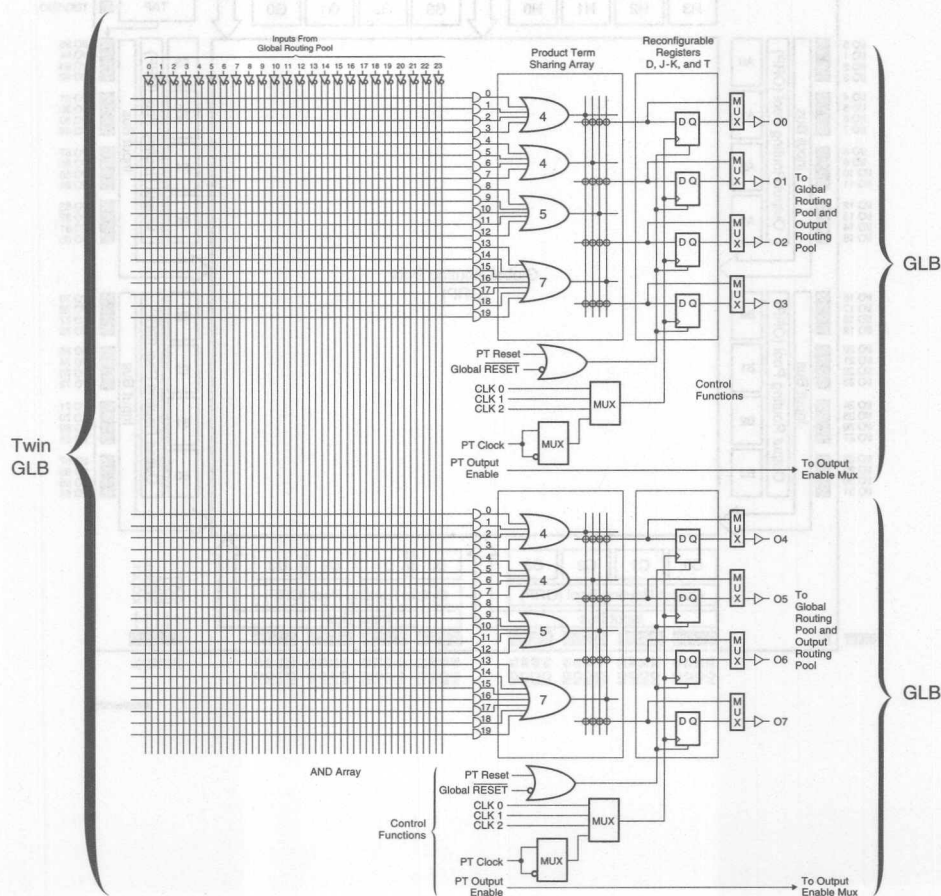
The Twin GLB is the standard logic block of the Lattice ispLSI and pLSI 3000 Family. This Twin GLB has 24 inputs, eight outputs and the logic necessary to implement most standard logic functions. The internal logic of the Twin GLB is divided into four separate sections: The AND Array, the Product Term Sharing Array, the Reconfigurable Registers, and the Control section.

The AND array consists of two 20 Product Term Sharing Arrays which can produce the logical sum of any of the 24 Twin GLB inputs. These inputs all come from the GRP, and are either feedback signals from any of the 32 Twin GLBs or inputs from the external I/O Cells. All Twin GLB

input signals are available to the Product Terms in both the logical true and complemented forms which makes Boolean logic reduction easier.

The two Product Term Sharing Arrays (PTSA) take the 20 Product Terms each and allocate them to four Twin GLB outputs. There are four OR gates, with four, four, five and seven inputs respectively. The output of any of these gates can be routed to any of the four Twin GLB outputs, and if more Product Terms are needed, the PTSA can combine them as necessary. If the user's main concern is speed, the PTSA can use a bypass circuit with four Product Terms to increase the performance of the cell. This can be done to any or all of the eight outputs of the Twin GLB.

Figure 2. Twin GLB: Product Term Sharing Array



3000 Family Architectural Description

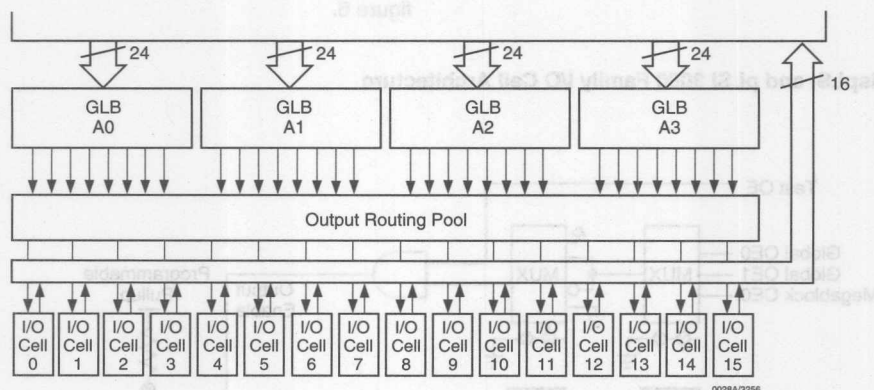
Megablock Structure

Four Twin GLBs make up a Megablock. Each GLB has a maximum fan-in of 24 inputs, and no dedicated inputs associated with any Megablock. A GLB has eight asso-

ciated outputs. A total of 32 GLB outputs are fed to the ORP. However, only 16 out of the 32 outputs feed to 16 I/O cells. The Megablock structure is shown in figure 3.

2

Figure 3. ispLSI and pLSI 3000 Family Megablock Block Diagram

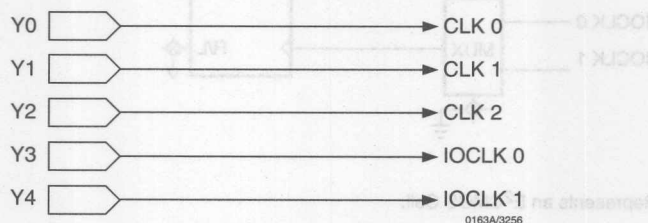


Global Clock Structure

The global clock structure is made up of five global clock input pins, Y0, Y1, Y2, Y3, and Y4. This is shown in figure 4. Three of the clock pins are dedicated for GLB clocks and the remaining two clock pins are dedicated for I/O register clocks. The clock GLB generation network which

is designed into the 1000 device family has been removed so all input clock signals are fed directly to the GLB clock input via a clock multiplexer. The GLB global clocks do not have inversion capability, but the product term clock does have inversion capability before it reaches the clock multiplexer.

Figure 4. ispLSI and pLSI 3000 Family Global Clock Structure



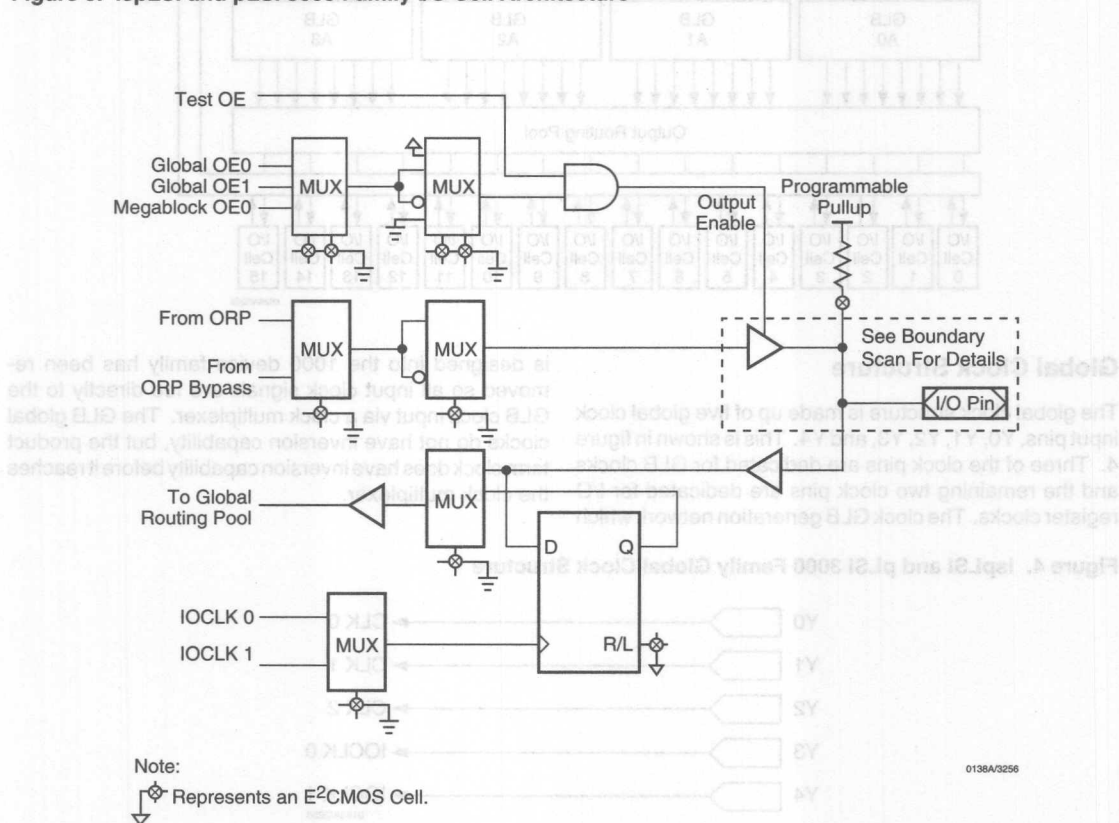
3000 Family Architectural Description

I/O Cells

The I/O cell structure architecture remains nearly the same as the 1000 Family as illustrated in figure 5. Each I/O cell now contains Boundary Scan Registers as shown in figure 8. An input pin has only one scan register as shown in figure 9. A global test OE signal is hardwired to

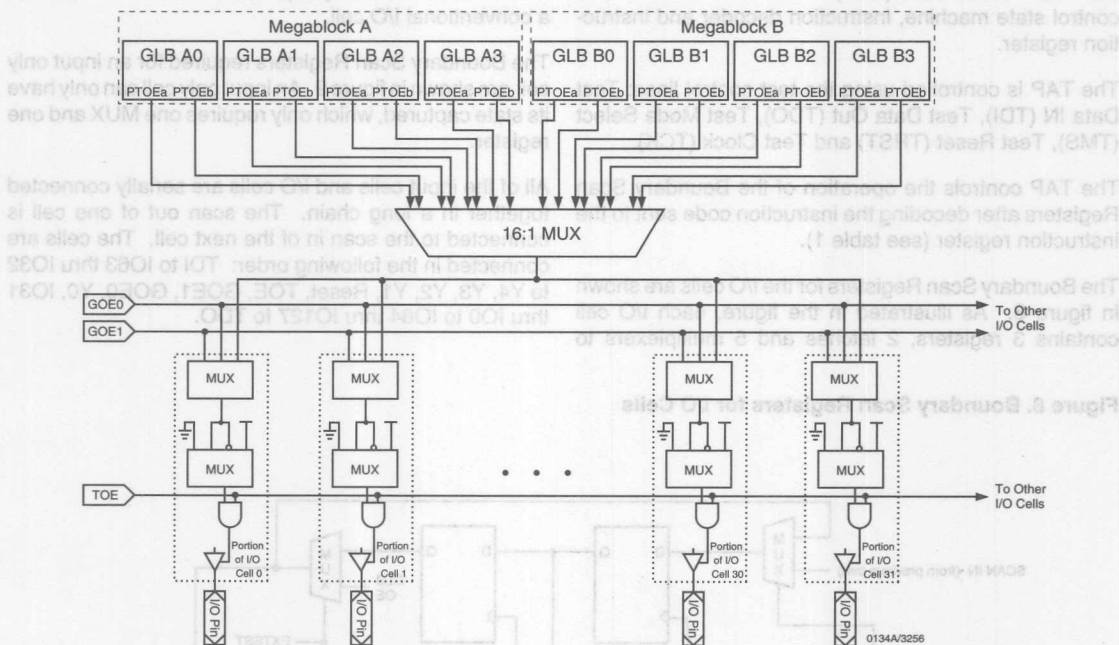
all I/O cells and is useful to perform static testing of all the 3-state output buffers within the device. In addition to the test OE signal, two global OEs are connected to all I/O pins. The product term OE is shared between two Megablocks resulting in twice the GLBs being able to use a single OE signal. The Megablock OE signal and global OE signals are fed to an OE multiplexer. The OE signals, with the exception of the test OE, have inversion capability after going through the OE multiplexer as shown in figure 6.

Figure 5. ispLSI and pLSI 3000 Family I/O Cell Architecture



3000 Family Architectural Description

Figure 6. ispLSI and pLSI 3000 Family Output Enable Controls

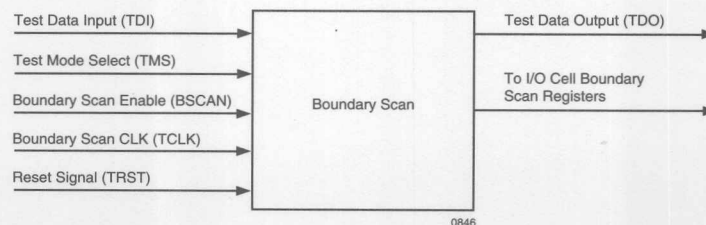


Boundary Scan

Boundary Scan (IEEE 1149.1 compatible) is a test feature incorporated within the device to provide on-chip test capabilities during PCB testing. Five input signal pins, BSCAN, TDI, TCLK, TMS, TRST, and one output signal pin, TDO, are associated with the boundary scan logic cells. These signal pins occupy the same dedicated signal pins used for ISP programming. The signal BSCAN is associated with the ispEN pin, TDI corresponds to the SDI pin, TCLK corresponds to the SCLK pin, TMS corre-

sponds to the MODE pin, and TDO corresponds to the SDO pin. When ispEN is asserted low, the MODE, SDI, SDO, and SCLK options become active for ISP programming. Otherwise, BSCAN, TDI, TCLK, TMS, TDO, and TRST options become active for boundary scan testing of the device. The boundary scan block diagram is shown in figure 7. TDI is the test data serial input, TCLK is the boundary scan clock associated with the serial shift register, TMS is the test mode select input, TDO is the test data output, and finally TRST is the reset signal pin.

Figure 7. Boundary Scan Block Diagram



3000 Family Architectural Description

The user interfaces to the boundary scan circuitry through the Test Access Port (TAP). The TAP consists of a control state machine, instruction decoder and instruction register.

The TAP is controlled using the test control lines: Test Data IN (TDI), Test Data Out (TDO), Test Mode Select (TMS), Test Reset (TRST) and Test Clock (TCK).

The TAP controls the operation of the Boundary Scan Registers after decoding the instruction code sent to the instruction register (see table 1).

The Boundary Scan Registers for the I/O cells are shown in figure 8. As illustrated in the figure, each I/O cell contains 3 registers, 2 latches and 5 multiplexers to

implement the ability to capture the state of the I/O cell or set the state of the output path of the cell or function as a conventional I/O cell.

The Boundary Scan Registers required for an input only cell are shown in figure 9. An input only cell can only have its state captured, which only requires one MUX and one register.

All of the input cells and I/O cells are serially connected together in a long chain. The scan out of one cell is connected to the scan in of the next cell. The cells are connected in the following order: TDI to IO63 thru IO32 to Y4, Y3, Y2, Y1, Reset, TOE, GOE1, GOE0, Y0, IO31 thru IO0 to IO64 thru IO127 to TDO.

Figure 8. Boundary Scan Registers for I/O Cells

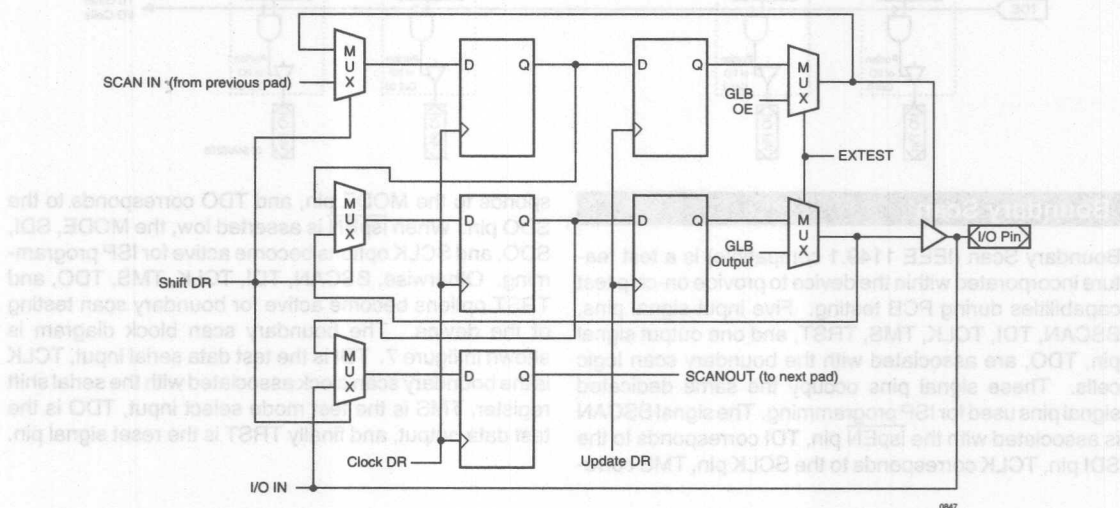


Figure 7. Boundary Scan Block Diagram



3000 Family Architectural Description

Figure 9. Boundary Scan Registers for an Input Only Cell

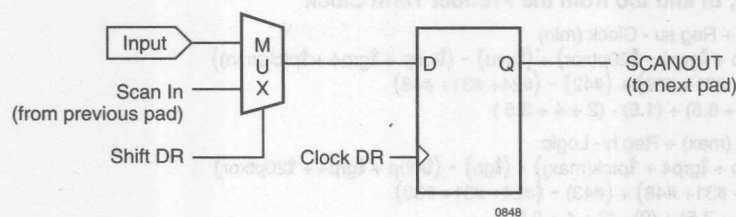


Table 1. Boundary Scan Instruction Codes

Instruction Name	Code	Description
SAMPLE/ PRELOAD	01	Loads and shifts data into BScan registers
EXTEST	00	Drives external I/O with BScan registers
BYPASS	11	Bypasses registers of selected device(s)

Note: MSB shifts in first.

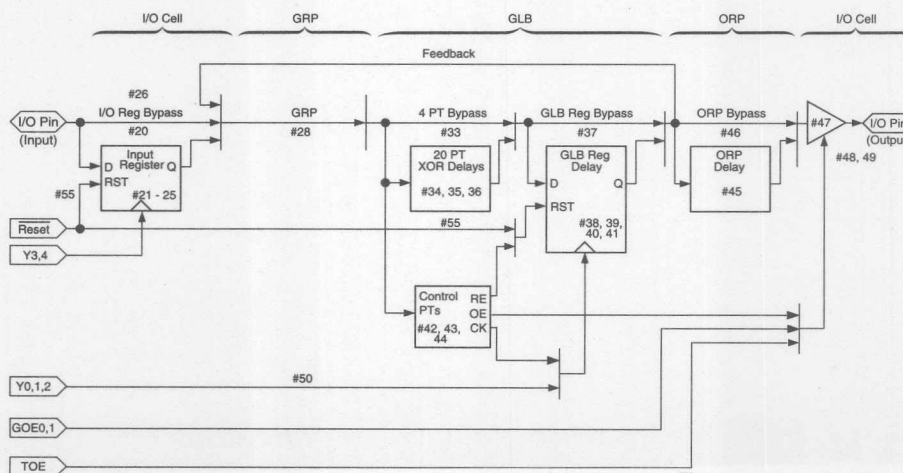
Table 10- 0006

Timing Model

The task of determining the timing through the device is simple and straightforward. A device timing model is shown in figure 10. To determine the time that it takes for data to propagate through the device, simply determine the path the data is expected to follow, and add the

various delays together (figure 11). Critical timing paths are shown in figure 10, using data sheet parameters. Note that the Internal timing parameters are given for reference only, and are not tested. (External timing parameters are tested and guaranteed on every device).

Figure 10. ispLSI and pLSI 3256 Timing Model



3000 Family Architectural Description

Figure 11. Timing Calculation Example

Derivations of t_{su} , t_h and t_{co} from the Product Term Clock¹

$$\begin{aligned}
 t_{su} &= \text{Logic} + \text{Reg su} - \text{Clock (min)} \\
 &= (t_{iobp} + t_{grp4} + t_{20ptxor}) + (t_{gsu}) - (t_{iobp} + t_{grp4} + t_{ptck(min)}) \\
 &= (\#24 + \#31 + \#39) + (\#42) - (\#24 + \#31 + \#48) \\
 6.5 \text{ ns} &= (2 + 4 + 8.5) + (1.5) - (2 + 4 + 3.5) \\
 t_h &= \text{Clock (max)} + \text{Reg h} - \text{Logic} \\
 &= (t_{iobp} + t_{grp4} + t_{ptck(max)}) + (t_{gh}) - (t_{iobp} + t_{grp4} + t_{20ptxor}) \\
 &= (\#24 + \#31 + \#48) + (\#43) - (\#24 + \#31 + \#39) \\
 8 \text{ ns} &= (2 + 4 + 7.5) + (9) - (2 + 4 + 8.5) \\
 t_{co} &= \text{Clock (max)} + \text{Reg co} + \text{Output} \\
 &= (t_{iobp} + t_{grp4} + t_{ptck(max)}) + (t_{gco}) + (t_{orp} + t_{ob}) \\
 &= (\#24 + \#31 + \#48) + (\#44) + (\#49 + \#51) \\
 20 \text{ ns} &= (2 + 4 + 7.5) + (1.5) + (2 + 3)
 \end{aligned}$$

1. Calculations are based upon timing specs for the ispLSI and pLSI 3256-70L

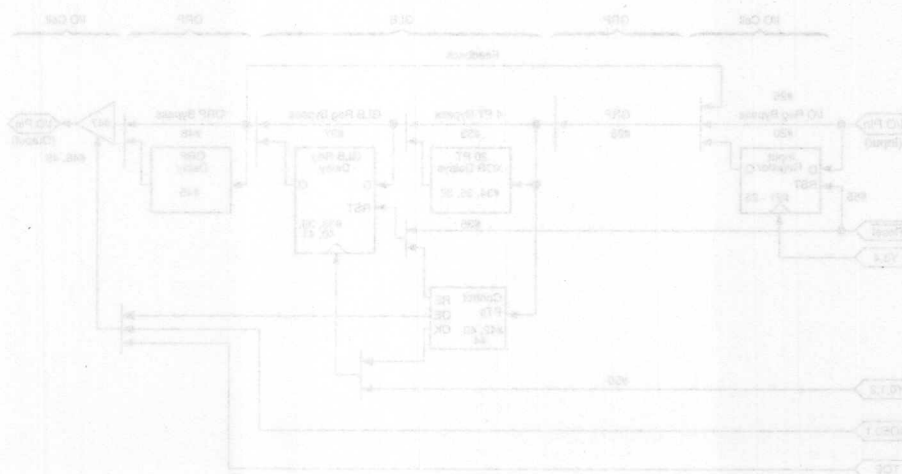
Instruction Name	Code	Description
SAMPLE PRELO	00	Drives external I/O with B-Scan registers
EXTST	01	Bypasses registers of selected device(s)
BYPASS	11	

Note: MSB shifts in first.

Timing Model

The task of determining the timing through the device is simple and straightforward. A device timing model is shown in figure 10. To determine the time that a takes for data to propagate through the device, simply determine the path the data is expected to follow, and add the various delays together (figure 11). Critical timing paths are shown in figure 10, using data sheet parameters. Note that the internal timing parameters are given for reference only and are not tested (External timing parameters are tested and guaranteed on every device).

Figure 10. ispLSI and pLSI 3256 Timing Model



ispLSI Architecture and Programming

2

ispLSI Programming Information

The following general programming information on the ispLSI (in-system programmable Large Scale Integration) devices describes how the internal state machine is implemented for programming and how to use the five programming interface signals to step through the state machine. The device specific information, such as timing and pin-outs, can be found in the Lattice Data Book. This section describes how to program ispLSI devices in a parallel configuration. For information on programming ispLSI devices in a serial daisy chain configuration please refer to the "Programming Multiple ISP Devices: Daisy Chain Configuration" Application Note located in Section 4 of this Handbook.

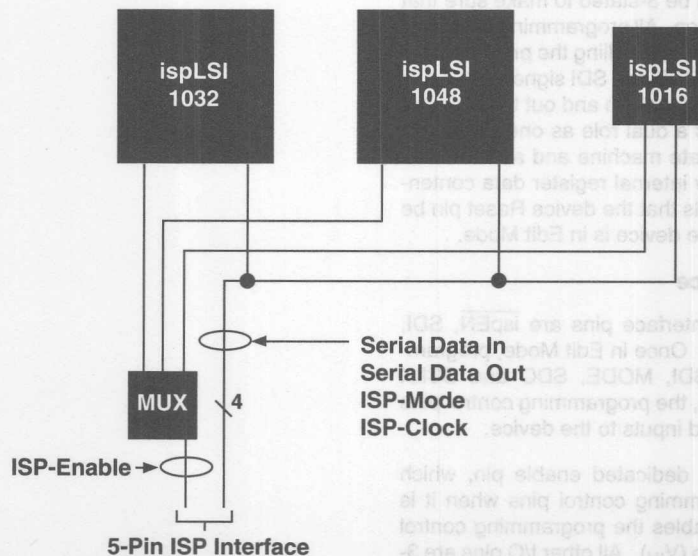
Programming Overview

To distinguish between normal operation and programming, two modes are defined: normal mode and edit mode. Once the device is in edit mode, the entire programming operation of the device is controlled by the

internal ISP state machine. The in-system programming enable (ispEN) signal controls the device operation modes.

The programming is controlled by the on-chip state machine via five programming interface signals. The ispEN signal is used to enable and disable the four programming control signals which include Serial Data In (SDI), Mode (MODE), Serial Data Out (SDO) and Serial Clock (SCLK). When the device is in normal mode, the four programming control signal pins can be used as normal Dedicated Input Pins. Figure 1 illustrates one such possible configuration for programming multiple ispLSI devices. With this scheme the ispEN signal for individual devices is enabled separately and one device is placed in the edit mode at a time. Since the other devices are in the normal mode, they can continue to perform normal system functions. This simple scheme requires connecting all four programming control signal pins together and precludes their use as dedicated inputs for normal system functions. ispEN is the only programming interface signal that is dedicated to a pin.

Figure 1. ispLSI Programming Interface



ispLSI Architecture and Programming

Normal Mode

In Normal Mode the four programming control pins become Dedicated Input pins. By multiplexing the programming control pins, these programming control pins can have a normal input function during Normal Mode. Figures 2 and 3 illustrate two alternate schemes which allow the designer to utilize the four programming control signal pins for performing normal system functions. Internal to the device, the programming functions are completely isolated from the normal operating functions when the device is in Normal Mode. Keeping the $\overline{\text{ispEN}}$ signal high puts the device in Normal Mode. For simplicity, the four programming control pins can be left unused for normal input functions. These pins can be reserved for ISP by using the ISP switch in the development tools. By leaving these pins unused, the programming interface is simplified when the programming signals and the Normal Mode input signals are not multiplexed.

Edit Mode

Programming circuitry is enabled by driving the $\overline{\text{ispEN}}$ signal low which puts the device in Edit Mode. In Edit Mode, all the functional I/O pins and input pins that are not used during programming are 3-stated. With the exception of the SDO signal, the remainder of the programming interface signals are input signals. When multiplexing the programming interface signals, the input driving the SDO pin must be 3-stated to make sure that there is no signal contention. All programming is accomplished in the Edit Mode by controlling the programming state machine with the MODE and SDI signals. SCLK is used to clock programming data in and out through SDI and SDO pins. SDI has a dual role as one of the two control signals for the state machine and as the serial data input. To avoid any internal register data contentions, Lattice recommends that the device Reset pin be pulled to ground when the device is in Edit Mode.

Programming Interface

The five programming interface pins are $\overline{\text{ispEN}}$, SDI, MODE, SDO and SCLK. Once in Edit Mode, programming is controlled by SDI, MODE, SDO and SCLK signals. In Normal Mode, the programming control pins can be used as dedicated inputs to the device.

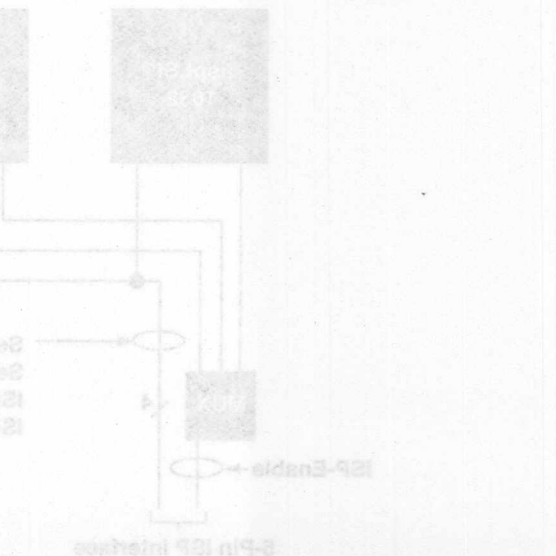
$\overline{\text{ispEN}}$ is an active low, dedicated enable pin, which enables the four programming control pins when it is driven low (V_{IL}) and disables the programming control pins when it is driven high (V_{IH}). All other I/O pins are 3-stated during Edit Mode and pulled up by the internal active pull-up resistors (equivalent to 100K Ω).

SDI performs two different functions. First, as the input to the serial shift register and second, as one of the two control pins for the programming state machine. Because of this dual role, SDI's function is controlled by the MODE signal. When MODE is low SDI is the serial input to the shift registers and when MODE is high SDI becomes the control signal. Internal to the device, the SDI is multiplexed to address shift register, high order data shift register and low order data shift register. The different shift instructions of the state machine determine which of these shift registers gets the input of the SDI.

The MODE signal combined with the SDI signal controls the programming state machine. This signal connects in parallel to all ispLSI devices.

SCLK is the serial shift register clock that is used to clock the internal serial shift registers. A low-to-high (positive) clock transition clocks the state machine. It also connects in parallel to all ispLSI devices. Similar to SDI, the shift instructions determine which of the shift registers are clocked for the data input from SDI.

SDO is the output of the serial shift registers. The selection of the shift register is determined by the state machine's shift instruction. In the flow through instruction and when MODE is driven high, SDO connects directly to SDI, and bypasses the device's shift registers. Since this is the only output pin for the Edit Mode, this signal will drive the external devices that are connected to this pin.



Programming Details

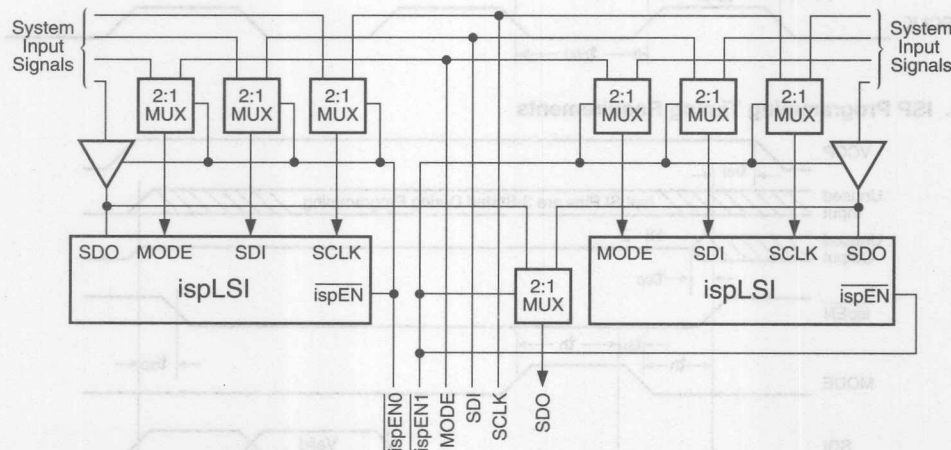
Programming is completely controlled by the state machine, once the device is in the Edit Mode. The state machine consists of three states, in which all programming related operations are performed. In order to run these programming operations, five bit instructions are defined (see table 2). Each instruction is then shifted into the device in one of the three states and executed in another state. The initial state of the state machine is used when the device is idle during edit, or to shift out the eight bit device identification code.

The following sections describe the general information about the critical timing parameters, state machine, state machine instructions, and device layout that apply to all the ispLSI devices. Any device specific information like the size of the shift registers and the device specific timing information can be found in the individual device data sheets.

There are various ways of programming the ispLSI devices. The easiest is to dedicate the ISP programming pins only for the programming functions. With dedicated ISP pins, one can program the devices in a parallel programming configuration (figure 1) where the programming signals are multiplexed. The parallel multiplexed programming method gives the user another advantage of being able to use the programming pins for system functions. Figure 2 illustrates a multiplexing scheme which allows the user to control the ISP programming through multiple $\overline{\text{ispEN}}$ signals. The multiple $\overline{\text{ispEN}}$ signals not only control the $\overline{\text{ispEN}}$ inputs of the ispLSI devices, but also is the control signal for multiplexing the functional signals and the ISP programming signals. The ISP programming signals MODE, SDI and SCLK function as inputs for normal functional mode as well as the ISP programming mode. SDO, however, functions as an input in normal functional mode and as an output in ISP programming mode. Figure 2 also shows the difference in controlling these different programming signals.

2

Figure 2. The Scan and Multiplex Programming Mode



ispLSI Architecture and Programming

Critical Timing Parameters

When programming ispLSI devices, there are several critical timing parameters that must be met to ensure proper programming. The two most critical parameters are the programming pulse width (t_{pwp}) and the bulk erase pulse width (t_{bew}). These pulse widths determine the programming and erasing of the E² cells. Figure 3 shows these critical program and erase timing specifications.

Along with the two programming and erasing specifications, the following timing specifications must also be met.

t_{isp} - Specifies the time it takes to get into the ISP mode after ispEN signal is activated or the time it takes to come out from the ISP mode after the ispEN becomes inactive.

t_{su} - Set up time of the control signals before the SCLK or the set up time of input signals against other control signals where applicable.

t_h - Hold time of the control signal after the SCLK. It also applies to the same input signals from the set up time.

t_{clkL} - Minimum clock pulse width.

t_{clkH}

t_{pww} - Verify or read pulse width. The minimum time requirement from the rising clock edge of verify/load instruction execution to the next rising clock edge (see figure 3).

t_{rst} - Power on reset timing requirement. t_{rst} must elapse after power up before any operations are performed on the device.

All the programming timing parameters are summarized in the timing diagram (see figure 4).

Figure 3. Program, Verify & Bulk Erase Timing

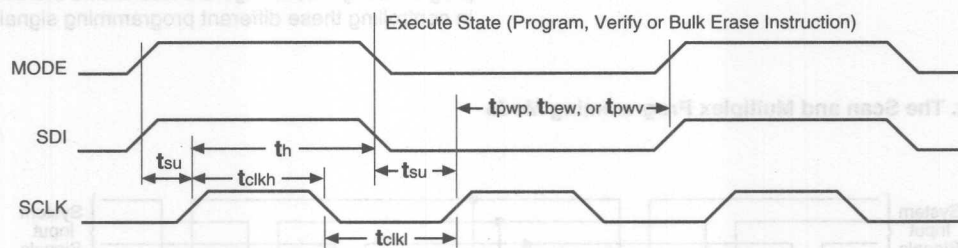


Figure 4. ISP Programming Timing Requirements

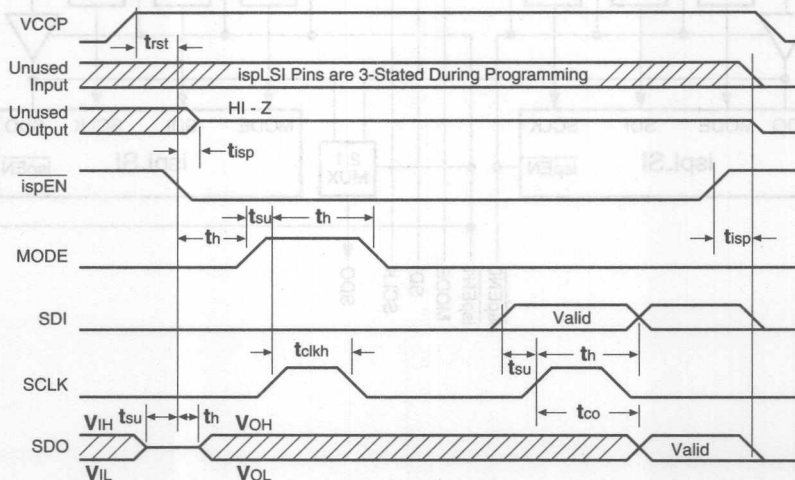
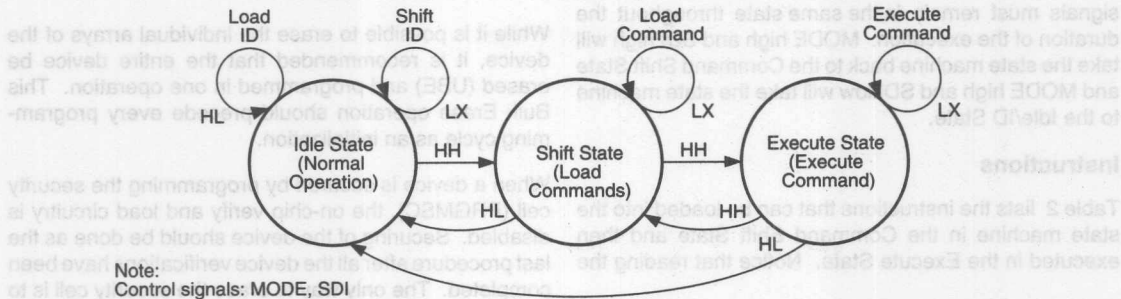


Figure 5. Programming State Machine



State Machine Operation

The state machine has three states to control programming and uses the MODE and SDI signals as inputs for each state. Based on these input signals, the state machine makes decisions to either stay in the same state or to branch to another state. The three states are Idle/ID State, Command Shift State and Execute State. The programming state machine diagram in figure 5 shows the three states and the logic levels of the control signals needed to force each transition state.

Idle/ID State

The Idle/ID state is the first state which is active when the device gets into the Edit Mode. The state machine is in the Idle/ID state when the device is idle, in the Edit Mode, or when the user needs to read the device identification. The eight bit device identification is loaded into the shift register by driving MODE high, SDI low and clocking the state machine with SCLK. Once the ID is loaded, it is read out serially by driving MODE low. Notice that when reading the device ID serially, SDI can either be high or low (don't care) and the state machine needs only seven clocks to read out eight bits of ID. The default state for the control signals is MODE high and SDI low. State transition to Command Shift State occurs when both MODE and SDI are high while state machine gets a clock transition. Table 1 lists the eight bit device ID's for all the ispLSI devices. As with most shift registers the Least Significant Bit (LSB) of the ID gets shifted out from the SDO first.

Command Shift State

This state is strictly used for shifting in the command instructions into the state machine. The entire five-bit instruction set is listed in the next section. When MODE is low and SDI is don't care in the Command Shift State,

Table 1. ispLSI Device ID Codes

Device	MSB	LSB
ispLSI 1016	00000001	
ispLSI 1024	00000010	
ispLSI 1032	00000011	
ispLSI 1048	00000100	
ispLSI 1048C	00000101	
ispLSI 2032	00010101	
ispLSI 3256	00100010	

SCLK shifts the instruction into the state machine. Once the instruction is shifted into the state machine, the state machine must transition to the Execute State to execute the instruction. Driving both MODE and SDI high and applying the clock will transfer the state machine from the Command Shift State to Execute State. If needed, the state machine can move from Command Shift State to Idle/ID State by driving MODE high and SDI low.

Execute State

In the Execute State, the state machine executes instructions that are loaded into the device in the Command Shift State. For some instructions, the state machine requires more than one clock to execute the command. An example of this multiple clock requirement is the address or data shift instruction. The number of clock pulses required for these instructions depends on the device shift register sizes (refer to the ISP programming section of the data sheet). When executing instructions such as Program, Verify or Bulk Erase, the necessary timing requirements must be followed to make sure that the commands are executed properly. For specific timing information refer to the individual data sheets.

ispLSI Architecture and Programming

To execute a command, the MODE is driven low and SDI is "don't care." For multiple clock instructions the control signals must remain in the same state throughout the duration of the execution. MODE high and SDI high will take the state machine back to the Command Shift State and MODE high and SDI low will take the state machine to the Idle/ID State.

Instructions

Table 2 lists the instructions that can be loaded into the state machine in the Command Shift State and then executed in the Execute State. Notice that reading the

device identification is done during the Idle/ID State and does not require an instruction.

While it is possible to erase the individual arrays of the device, it is recommended that the entire device be erased (UBE) and programmed in one operation. This Bulk Erase operation should precede every programming cycle as an initialization.

When a device is secured by programming the security cell (PRGMSC), the on-chip verify and load circuitry is disabled. Securing of the device should be done as the last procedure after all the device verifications have been completed. The only way to erase the security cell is to perform a bulk erase on the device.

Table 2. State Machine Instruction Set

Instruction	Operation	Description
00000	NOP	No operation performed
00001	ADDSHFT	Address Register Shift: Shifts address into the address shift register from SDIN.
00010	DATASHFT	Data Register Shift: Shifts data into or out of the data serial shift register.
00011	UBE	User Bulk Erase: Erase the entire device.
00100	GRPBE	Global Routing Pool Bulk Erase: Bulk erases the GRP array only.
00101	GLBBE	Generic Logic Block Bulk Erase: Bulk erases all the GLB array only.
00110	ARCHBE	Architecture Bulk Erase: Bulk erases the architecture array and I/O configuration only.
00111	PRGMH	Program High Order Bits: The data in the data shift register is programmed into the addressed row's high order bits.
01000	PRGML	Program Low Order Bits: The data in the data shift register is programmed into the addressed row's low order bits.
01001	PRGMSC	Program Security Cell: Programs the security cell of the device.
01010	VER/LDH	Verify/Load High Order Bits: Load the data from the selected row's high order bits into the data shift register for verification.
01011	VER/LDL	Verify/Load Low Order Bits: Load the data from the selected row's low order bits into the data shift register for verification.
01100	GLBPRLD	Generic Logic Block Preload: Preloads the registers in the GLB with the data from SDIN. All registers in the GLB form a serial shift register. Refer to device layout section for details.
01101	IOPRLD	I/O Preload: Preloads the I/O registers with the data from SDIN. All registers in the I/O cell form a serial shift register (the same order as GLB registers).
01110	FLOWTHRU	Flow Through: Bypasses all the internal shift registers and SDOOUT becomes the same as SDIN.
10010	VE/LDH	Verify Erase/Load High Order Bits: Load the data from the selected row's high order bits into the data shift register for erased verification.
10011	VE/LDL	Verify Erase/Load Low Order Bits: Load the data from the selected row's low order bits into the data shift register for erased verification.

Device Layout

The purpose of knowing the device layout is to be able to translate the JEDEC format programming file into the serial data stream format for programming ispLSI devices. Two main factors determine how the translation is implemented: the length of the address shift register and the length of the data shift register. The length of the address shift register indicates how many rows of data are to be programmed into the device. The length of the data shift register indicates how many bits are to be programmed in each row. Both registers operate on the First In First Out (FIFO) basis where the Least Significant Bit (LSB) of the data or address is shifted in first and the Most Significant Bit (MSB) of the data or address is shifted in last. For the data shift register, the low order bits and the high order bits are separately shifted.

Each ispLSI device has a predefined number of address rows and data bits needed to access its E²CMOS® cells during programming. The data bits span the columns of the E² array. From this information the number of programming cells (or fuses) are determined. Table 3 highlights the address and data shift register (SR) sizes for all ispLSI devices. The JEDEC file for these ispLSI devices will reflect the number of cells (fuses) seen in table 3. The total number of cells becomes critical if the programming patterns are to be stored in an on-board memory storage of limited capacity such as EPROM or PROM.

The L-field in the JEDEC programming file indicates the first cell number of each row. The JEDEC standard requires that there is at least the beginning cell number L00000. L-fields of the subsequent lines are optional. From this reference cell location all other cell locations can be determined. Zero in the cell location indicates that the E² cell in that particular location is programmed (or has a logic connection equivalent to a metal fuse being intact). A one (1) in the cell location indicates that the cell is erased (equivalent to a blown fuse). The fusemap operation in the Lattice software generates this JEDEC standard programming file.

Fuse Map to Device Conversion

One of the major elements needed to program an ispLSI device is the JEDEC fuse map in which the specific logic implementation is stored. While the ispCODE software takes care of these details, it is important to understand how this JEDEC fuse map is mapped onto the physical ispLSI device during programming. The physical layout of the fuse pattern begins with Address Row 0 and ends with the maximum Address Row N and is determined by the length of the Address SR as described in table 3. Spanning the Address Rows are the outputs of the High-Order Data SR and Low-Order Data SR, as described in table 4. Programming fuses on a given row are enabled by a "1" within the Address Shift Register for the appropriate row and the use of state machine instructions that selectively operate on the High-Order Data SR or the Low-Order Data SR. For example, the PRGMH instruction programs the High-Order data bits within the device for the selected Address Row and the PRGML instruction programs the Low-Order data bits (table 2 lists the ISP state machine instructions). Referring to figure 6, the starting cell (L00000) of the JEDEC fuse map shifts into the device at the physical location corresponding to Address Row 0, High-Order Data SR bit 0. n and m in the figure refer to the Address SR length and the Data SR length, respectively, of the device (refer to table 3). A series of sequential shifts eventually results in the last cell location (Total # of Cells - 1) of the JEDEC fuse map shifting into Address Row (n-1), Low-Order Data SR bit (m-1) on the actual device.

The ispCODE Software routines make use of a bit packed data format, called ispSTREAM™, to transfer data between the JEDEC fuse map and the physical device locations. The JEDEC fuse map can be translated into ispSTREAM using the `isp_jedtoisp` function and the ispSTREAM format can be translated into a JEDEC fuse map using the `isp_isptofjed` function.

Command Stream

The first step of programming the ispLSI devices is to determine the type of device to be programmed. This can be done by reading the eight-bit device ID of all the

Table 3. ispLSI Address and Data Shift Register and Total Cell Summary

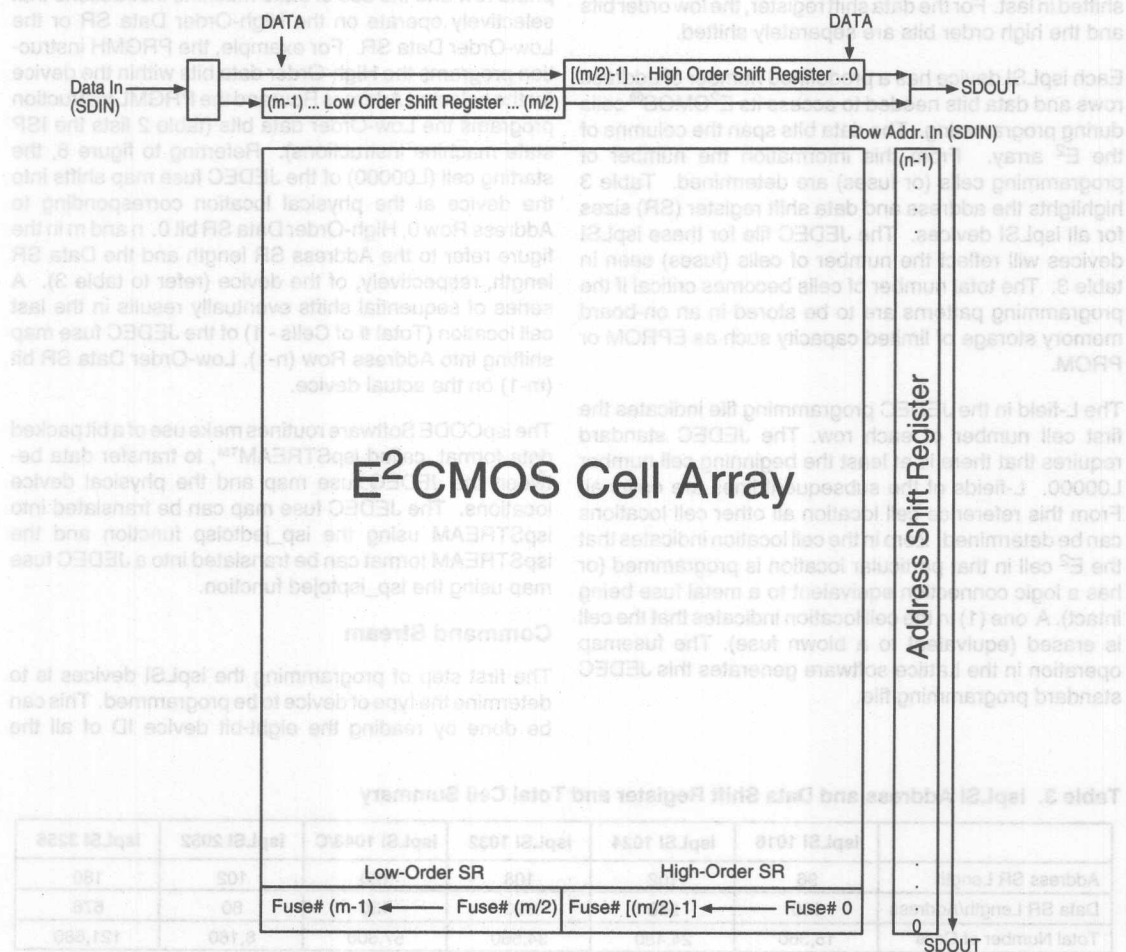
	ispLSI 1016	ispLSI 1024	ispLSI 1032	ispLSI 1048/C	ispLSI 2032	ispLSI 3256
Address SR Length	96	102	108	120	102	180
Data SR Length/Address	160	240	320	480	80	676
Total Number of Cells	15,360	24,480	34,560	57,600	8,160	121,680

ispLSI Architecture and Programming

Table 4. Summary of Data Shift Register Bits

Data SR Bits	ispLSI 1016	ispLSI 1024	ispLSI 1032	ispLSI 1048
High Order Data SR LSB	0	0	0	0
High Order Data SR MSB	79	119	159	239
Low Order Data SR LSB	80	120	160	240
Low Order Data SR MSB	159	239	319	479
Data SR Size (Bits)	160	240	320	480

Figure 6. ispLSI Device to Fuse Map Translation



devices. By keeping the SDI to a known level (either high or low), the ID shift can be terminated when a sequence of eight ones or eight zeros is read. From the device ID the serial bit stream for programming can be arranged. A typical programming sequence is as follows:

- 1) ADDSHFT command shift
- 2) Execute ADDSHFT command
- 3) Shift address
- 4) DATASHFT command shift
- 5) Execute DATASHFT command
- 6) Shift high order data
- 7) PRGMH command shift
- 8) Execute PRGMH
- 9) DATASHFT command shift
- 10) Execute DATASHFT command
- 11) Shift low order data
- 12) PRGML command shift
- 13) Execute PRGML
- 14) Repeat from 1) until all rows are programmed.

Diagnostic Register Preload

This section explains how to preload all of the buried registers and I/O registers to a known state to test the logic function of a device. The process of loading the register will reduce the time necessary to test a function that is deeply embedded in the logic of an ispLSI device.

To preload a device the ISP state machine is used with the same five pins that are used for programming ispEN, SDI, MODE, SDO and SCLK. Two state machine commands preload all of the registers: GLBPRLD and

IOPRLD. These two commands enable two different shift registers and enable data to be loaded into the device. The process of loading data into the device is:

1. Enter the ISP programming mode by driving ispEN pin to V_{il}.
2. Load command GLBPRLD and execute command (wait one t_{clk}).
3. Clock in the GLB preload data.
4. Load the command IOPRLD and execute the command (wait one t_{clk}).
5. Clock in the I/O preload data.
6. Return to the normal mode by driving the ispEN pin to V_{ih}.
7. Execute the vectors.

When preloading a device it is important to keep the dedicated input pins (RESET, Y0, Y1, Y2 and Y3) in the same state as the previous vector. If the state of these pins is switched during the preload sequence the register may not load correctly and the results cannot be guaranteed.

The preload feature is not recommended on designs which use product term resets. The asynchronous nature of these resets can cause registers to be reset unexpectedly, therefore the results cannot be guaranteed.

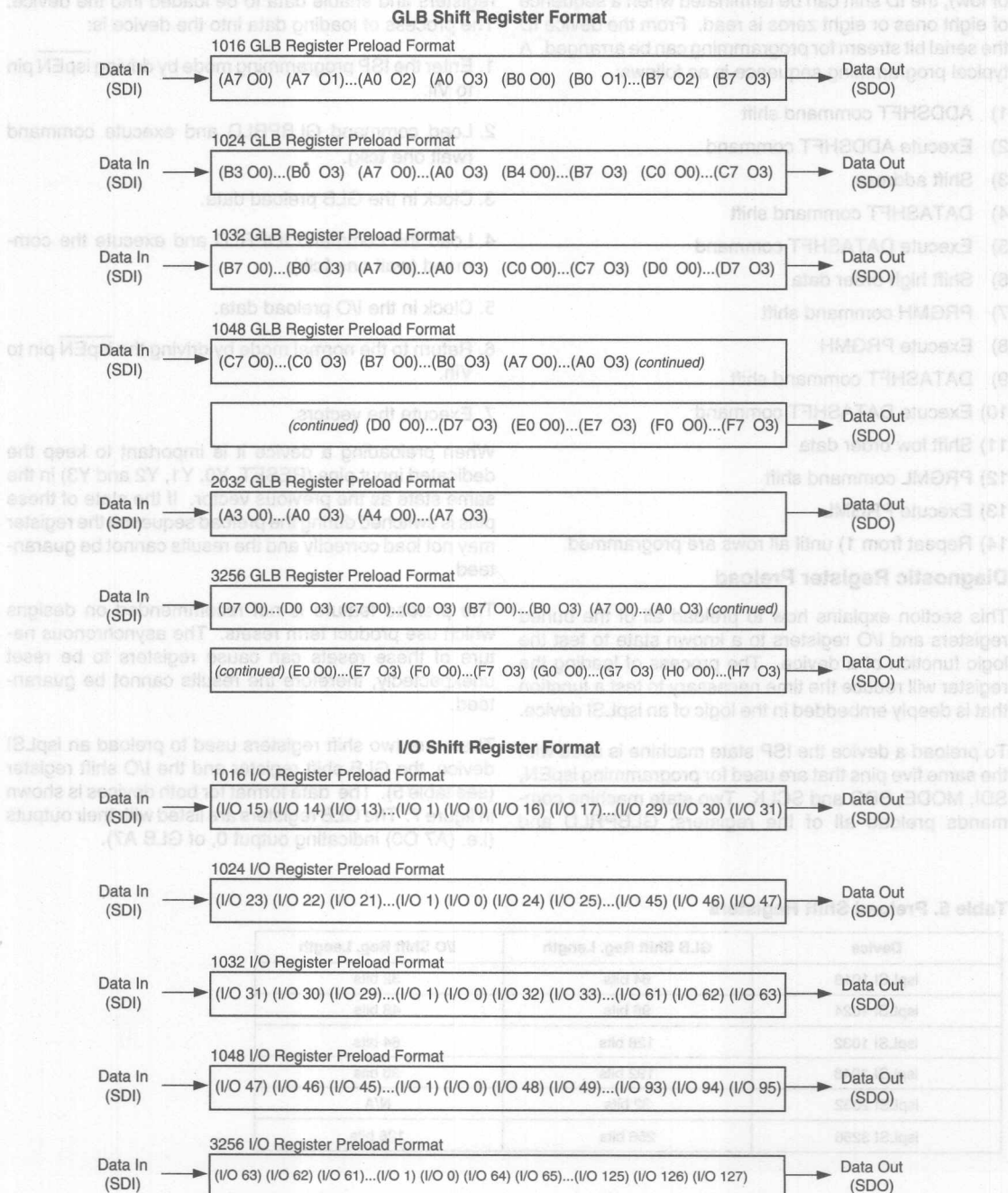
There are two shift registers used to preload an ispLSI device, the GLB shift register and the I/O shift register (see table 5). The data format for both devices is shown in figure 7. The GLB registers are listed with their outputs (i.e. (A7 O0) indicating output 0, of GLB A7).

Table 5. Preload Shift Registers

Device	GLB Shift Reg. Length	I/O Shift Reg. Length
ispLSI 1016	64 bits	32 bits
ispLSI 1024	96 bits	48 bits
ispLSI 1032	128 bits	64 bits
ispLSI 1048	192 bits	96 bits
ispLSI 2032	32 bits	N/A
ispLSI 3256	256 bits	128 bits

ispLSI Architecture and Programming

Figure 7. GLB Shift Register and I/O Shift Register Format



ISP Programming Support

To assist users in implementing the ISP programming, Lattice provides the isp Engineering Kit hardware and ispCODE C language software routines which implement the basic ISP functions for programming. The Lattice ISP programming support uses the PC parallel port to program the devices.

isp Engineering Kit Hardware Definition

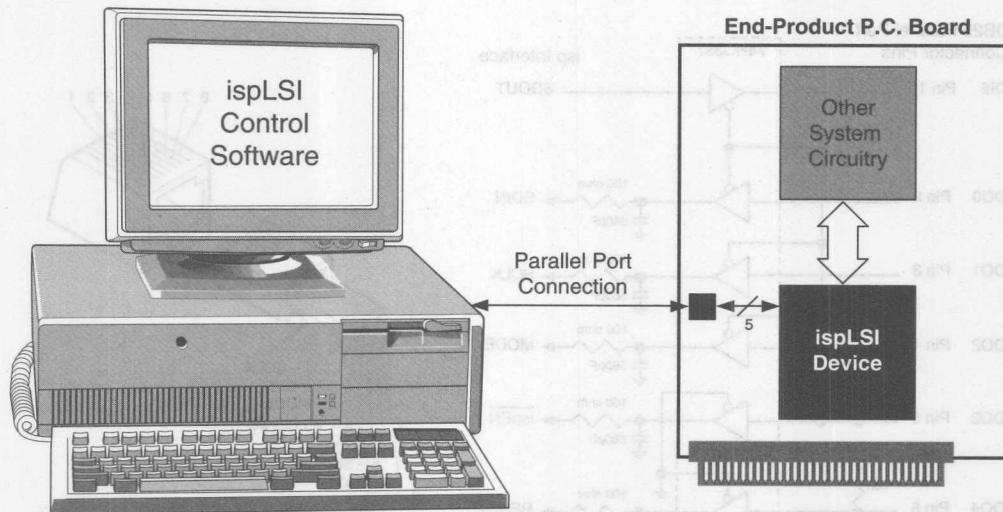
Lattice provides both a PC-based (Model 100) and a Sun Workstation-based (Model 200) isp Engineering Kit. PC-based, parallel I/O port programming interface implementation is explained in this section. For details on the Model 200 refer to the Model 200 isp Engineering Kit datasheet. The main function of this ispLSI programming interface is to provide four properly timed programming signals and the ispEN signal to the device. The PC parallel port is used in the isp Engineering Kit to provide these programming signals. The signals driven by the parallel port can be used either by the Lattice isp Programming Module (part of isp Engineering Kit Model

100) or on the system board if the circuit board is built with provisions to connect the ISP programming signals to the appropriate traces.

In the case of users using the PC serial port as the I/O port for programming, the serial data must be converted by additional circuitry into the appropriate programming signals. There must also be timing circuitry that translates the serial instructions into timed ISP programming signals. This section only discuss the parallel port interface. Lattice's isp Engineering Kit Model 200 supports serial port programming.

In order to use the PC parallel port, the parallel port operation must be defined properly. After defining the port, it is just a matter of developing the programming software to read and write from the parallel port. To guarantee the signal integrity and drive capability, a 74HC367 buffer should be directly connected at the parallel port's DB25 connector. Figure 9 defines the parallel port DB25 pins and the associated ISP programming signals. The global RESET signal is also provided to ensure a proper register reset after programming.

Figure 8. Configuring an ispLSI Device from a Remote System



ispLSI Architecture and Programming

The buffer then drives the cable that connects the output of the buffer to the ISP pins of the device. It is important to keep the cable length to a minimum to reduce the loading on the signal drivers. Since ispEN , SDI, SCLK and MODE are inputs to the ispLSI device, they are being driven by the buffer connected to the parallel port. SDO, on the other hand, is an output signal which the ispLSI device has to drive. If the load on the SDO signal is more than a minimum length cable and the parallel port input, it is recommended that the user provide a buffer on the circuit board to ensure signal integrity.

For the parallel port interface, the software must access the proper parallel port address. Once the port is defined, the data transfer is accomplished by simply reading from the port and writing to the port. The software must also guarantee proper timing between the ISP programming signals. When the programming software is executed, most of the shorter hardware timing requirements are automatically met due to the relatively long instruction execution times. The programming pulse width (tpwp) and bulk erase pulse width (tbew), which are in the 40ms to 200ms range, are the hardware timings that typically require wait states in the software. The example functions in the ispCODE illustrates reading of the computer's timer chip to generate these wait states.

Based on the programming pulse width requirement, the total programming time can be estimated. Since the shifting the address and data is relatively small compared to the programming time, the total programming time can be estimated by the following formula.

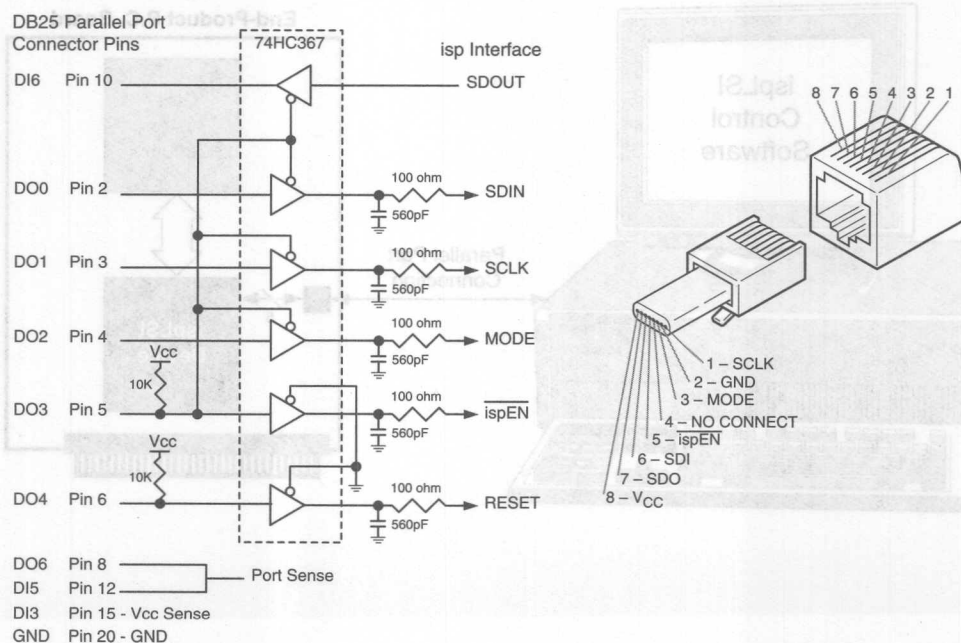
$$\text{Total Programming Time} = \text{Address SR Length} \times 2 \times \text{tpwp}$$

Assuming that the programming pulse width (tpwp) is 100ms, the total programming time for the ispLSI 1048 is approximately 24 seconds.

Microprocessor-Based Programming

Similar to PC-based I/O port controlled programming, a processor or a microprocessor can be used to directly supply the ISP programming signals with minimum decoding logic and an optional storage device (see figure 10). The discussion in this section pertains to the implementation of ISP programming on a circuit board with a microprocessor. The discussion is based on the assumption that the patterns and the code are stored in EPROMs. Since an efficient use of storage is desirable, the bit packed ispSTREAM format will use the least amount of storage. The basic requirement here, again, is to supply properly timed ISP programming signals.

Figure 9. PC Parallel Port Buffer & RJ45 Connector Definition



Hardware Configuration

There are several ways to define the ISP programming hardware depending on the type of storage device and how the ispLSI devices are to be programmed. The hardware configuration shown in figure 11 uses an 8-bit wide EPROM to store the fuse maps and code. The patterns are then read from the EPROM by the microprocessor and converted into serial stream format. The ISP signals are driven from the decoder and I/O port which decodes the proper ISP read/write address space similar to the I/O port definition of the previous setup. Similarly, fuse map memory addresses must also be defined to be properly read from the EPROM.

Programming pattern storage requirements are directly dependent upon the ispLSI device type and which ISP functions must be executed by the microprocessor. Assuming the bit packed ispSTREAM format for the fuse map, the number of bytes required for each ispLSI device is simply the total number of cells divided by eight. In the case of ispLSI 1048, 7.2K bytes is required to store the JEDEC fuse map.

Similar to the parallel port interface, most hardware timing requirements can be satisfied by the software instruction execution time. Only the program, verify and bulk erase times requires the software to have wait cycles. Many microprocessor boards will not have a timer chip to time the wait states. However, the instruction execution times typically can be accurately estimated. Therefore, timing loops must be inserted into the instructions control critical hardware timing.

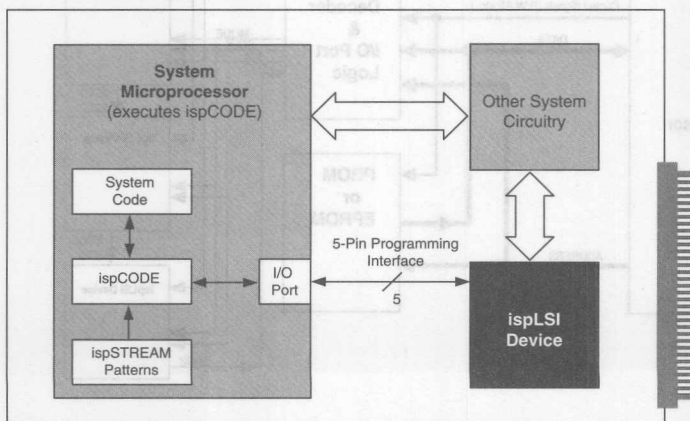
Software support for this case is very similar to the previous. Within the software, however, address spaces for the ISP read/write locations and the EPROM read locations must be defined. The storage space requirement for the code must also be determined if the code is going to reside in the storage device. Based on the ispCODE functions, the object code which is capable of executing basic ISP functions typically does not exceed 8K byte of memory. This memory requirement is directly proportional to the amount of ISP and user interface functions.

ISP Software Interface

In addition to the hardware interface, the ispCODE C language routines take care of the ispLSI programming software interface. The software interface must implement routines to read and write from the parallel port, to translate the JEDEC fusemap to and from the stream file format, and to toggle the ISP hardware signals connected at the output port. Predefined routines for these functions such as `isp_program`, `isp_read`, `isp_verify`, etc. are provided with the ispCODE. The ispLSI user only needs to integrate these routines into their overall system software.

The ispCODE routines makes use of the ispSTREAM bit packed data format to transfer data between the JEDEC fuse map and the physical device location. The JEDEC fuse map can be translated into ispSTREAM using the `isp_jedtoisp` function and the ispSTREAM format can be translated into a JEDEC fuse map using the `isp_isptojed` function. In addition to the fuse map translation routines, the ispCODE provides functions to check the device ID, to read and write the User Electronic Signature (UES),

Figure 10. Configuring an ispLSI Device from an On-Board Microprocessor



ispLSI Architecture and Programming

and to keep track of the program cycle counter. Refer to the ispCODE User Manual for more details.

ispLSI Device Special Features

In addition to transferring the fuse pattern into the ispLSI device with proper ISP timing, there are a few administrative functions that can make device programming more efficient when implemented in the ISP programming algorithm.

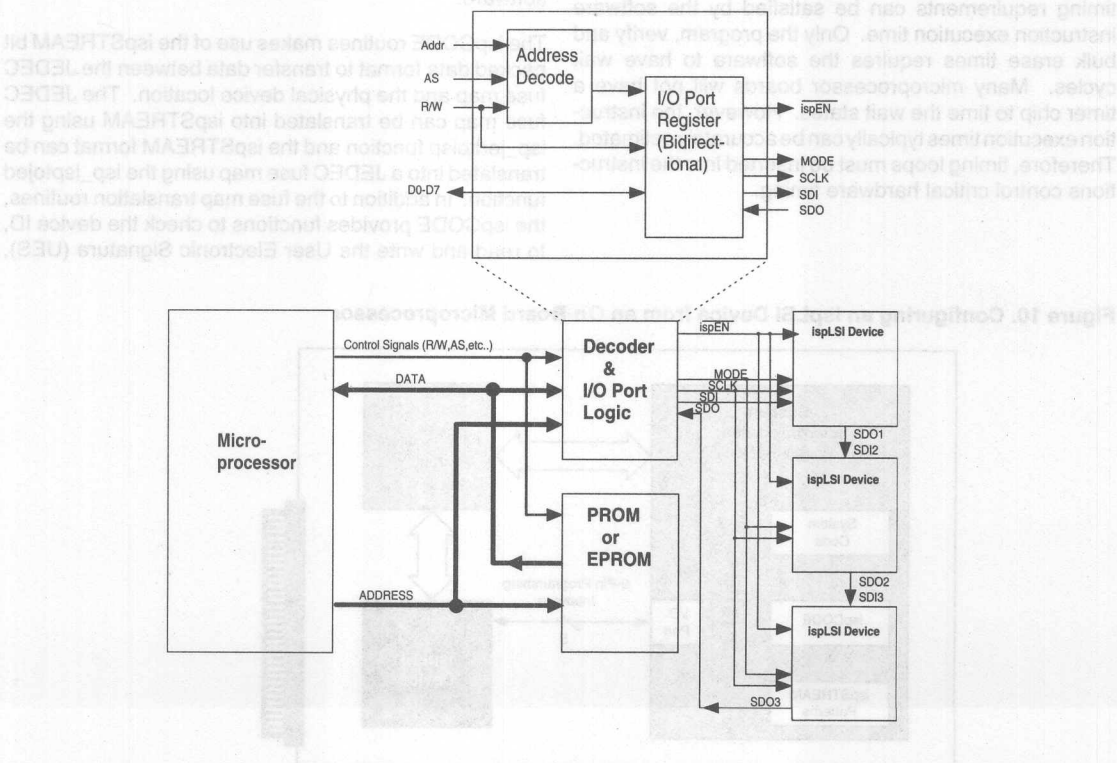
All ispLSI devices have hardwired device identification codes. These ID codes should be used to identify proper device and fuse map compatibility. The ID check should be run as the very first procedure before any device programming procedures. The ispCODE routines provided by Lattice include the `isp_get_id` function to facilitate this process.

The ispLSI devices also provide several programmable locations for the UES and program cycle counter. The UES can be used to identify which pattern is programmed

into the device. This is a very useful way of electronically identifying the devices and their programs, especially when the devices are secured. A 16-bit program cycle counter can be implemented within the reserved location, similar to the UES, to keep track of the number of program cycles which the device experiences to avoid exceeding the maximum programming cycle limit. UES and program cycle counter routines are provided as part of Lattice's ispCODE software.

One of the diagnostic features of the ispLSI devices is register preload. GLB and I/O registers become serial shift registers during the register preload command execution. Data can either be shifted into or out of these shift registers for system diagnostic functions. Special attention must be paid to the GLB and I/O clocks in order to use the register preload features properly. One must drive all GLB or all I/O clocks high throughout the execution of the GLB or I/O preload commands. This means that when defining the test pattern that uses the preload commands all GLB or all I/O clock polarities must be the same.

Figure 11. Microprocessor Board Configuration



Boundary Scan

The Lattice 3000 family of devices supports the IEEE 1149.1 Boundary Scan specifications. The following sections explain in detail how to interface to the devices through the Test Access Port (TAP), how the boundary scan registers are implemented within the devices, and the boundary scan instructions that are supported by the ispLSI and pLSI 3000 family.

Test Access Port (TAP)

The test access port of the boundary scan is accessed through six interface signals. These interface signals have dual functions in the case of ispLSI 3000 family and are used for Boundary Scan interface and in-system programming interface signals. For the pLSI 3000 family the six interface signals are only used for the boundary scan TAP interface. Table 6 describes the interface signals.

The above mentioned six signals are dedicated for Boundary Scan use for the pLSI family of devices. Since ISP programming is accomplished through the same pins, five of the six signals have both Boundary Scan interface and ISP functions on the ispLSI devices. The TRST is the only signal that does not have a dual function. It is only used to reset the TAP controller state machine. The sequencing of test routines are governed by the TAP controller state machine. The state machine uses the TMS and TCK signals as its inputs to sequence the states. Figure 12A is the IEEE1149.1 specified state machine where the condition for the state transition is the state of the TMS input condition before TCK within a

given state. The timing specification is also shown on figure 12B.

The main features of the TAP controller state machine consists of Test-Logic-Reset state to reset the controller and the Run-Test states. Two main components of the Run-Test states are Data Register (DR) control states and Instruction Register (IR) control states. Both of these register control states are organized in a similar manner where one can capture the registers, shift the register string, or update the registers. Capturing the DRs simply loads the DR with the data from the corresponding functional input, output, or I/O pins. The IR capture, on the other hand, loads the IRs with the previously executed instruction bits. Shift register states serially shifts the DR and IR. In the case of DR shift, the data is shifted according to the order of the inputs, outputs, and I/Os defined in the Boundary Scan section of each device data sheet. The IRs are shifted out from the least significant bit first. During update registers states, the DRs update the latches to drive the external pins and IRs update the instruction bits with the instruction that is to be executed.

Boundary Scan Registers

In order to support Boundary Scan, two types of data registers are defined for the ispLSI and pLSI devices — I/O cell registers and input cell registers. The main purpose of these registers is to capture test data from the appropriate signals and shift data to either drive the test pins or examine captured test data.

Figure 13 describes the register for the I/O cell. The I/O cell, by definition, must have three components. One register component captures the output enable (OE) signal, the second component captures the output data

Table 6. Boundary Scan Interface Signals

pLSI 3000 Family	ispLSI 3000 Family	Pin Function Description
BSCAN	BSCAN/ispEN	Active high signal on this pin selects the Boundary Scan function while active low signal selects the ISP function on the ispLSI devices. Internal pullup on this pin drives the signal high when the external pin is not driven.
TCK	TCK/SCLK	Test Clock function for Boundary Scan and serial clock for the ISP function.
TMS	TMS/MODE	Test Mode Select for Boundary Scan and MODE control for ISP function.
TDI	TDI/SDI	Test Data Input for Boundary Scan and Serial Data Input for ISP pin functions as serial data input pin for both interfaces.
TRST	TRST	Test Reset Input is an asynchronous signal to initialize the TAP controller to Test-Logic-Reset state.
TDO	TDO/SDO	Test Data Output for Boundary Scan and Serial Data Output for ISP pin functions as serial data output pin for both interfaces.

ispLSI Architecture and Programming

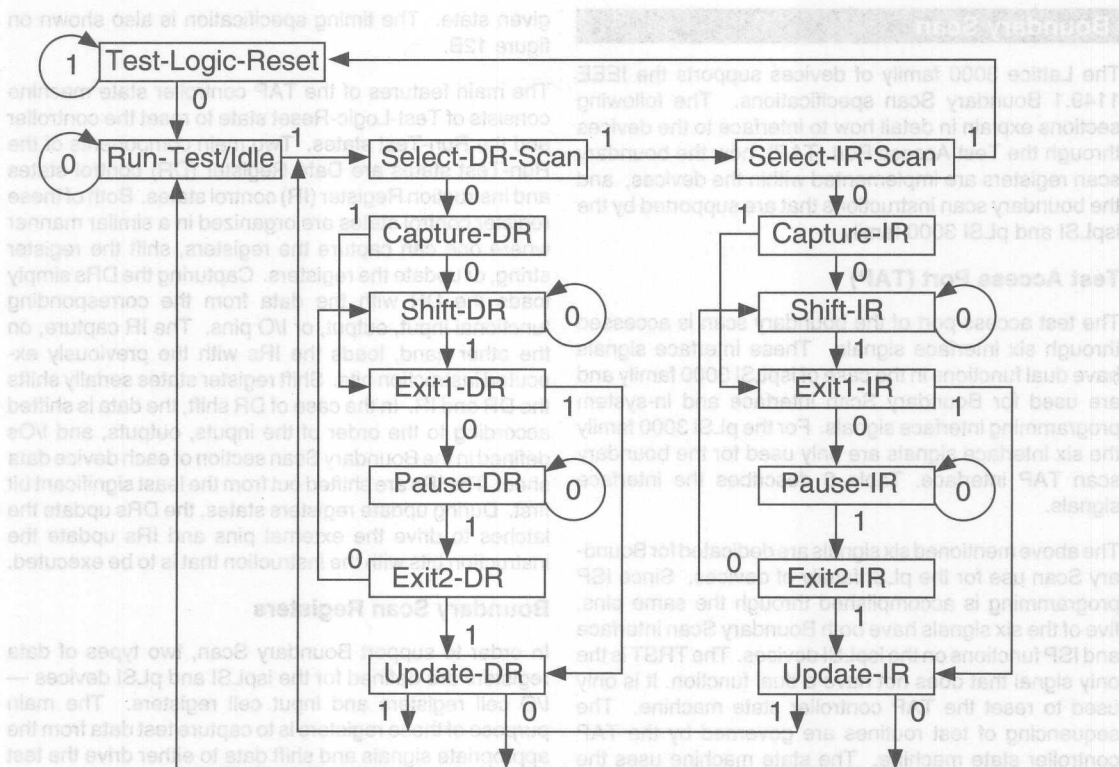


Figure 12A. TAP Controller State Machine

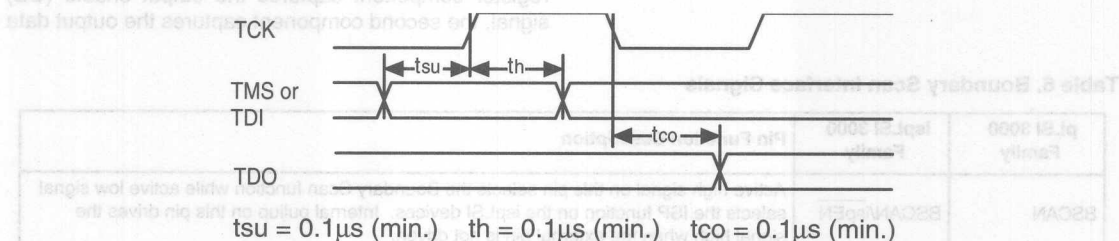


Figure 12B. TAP Controller Timing Diagram

TDO	Test Data Output for Boundary Scan and Serial Data Output for ISP pin functions as serial data input pin for both interfaces.
TRST	Test Reset Input is an asynchronous signal to initialize the TAP controller to Test-Logic-Reset state.
TDI	Test Data Input for Boundary Scan and Serial Data Input for ISP pin functions as serial data input pin for both interfaces.
TMS	Test Mode Select for Boundary Scan and Mode Select for ISP functions.
TCK	Test Clock Input for Boundary Scan and Serial Clock for ISP functions.

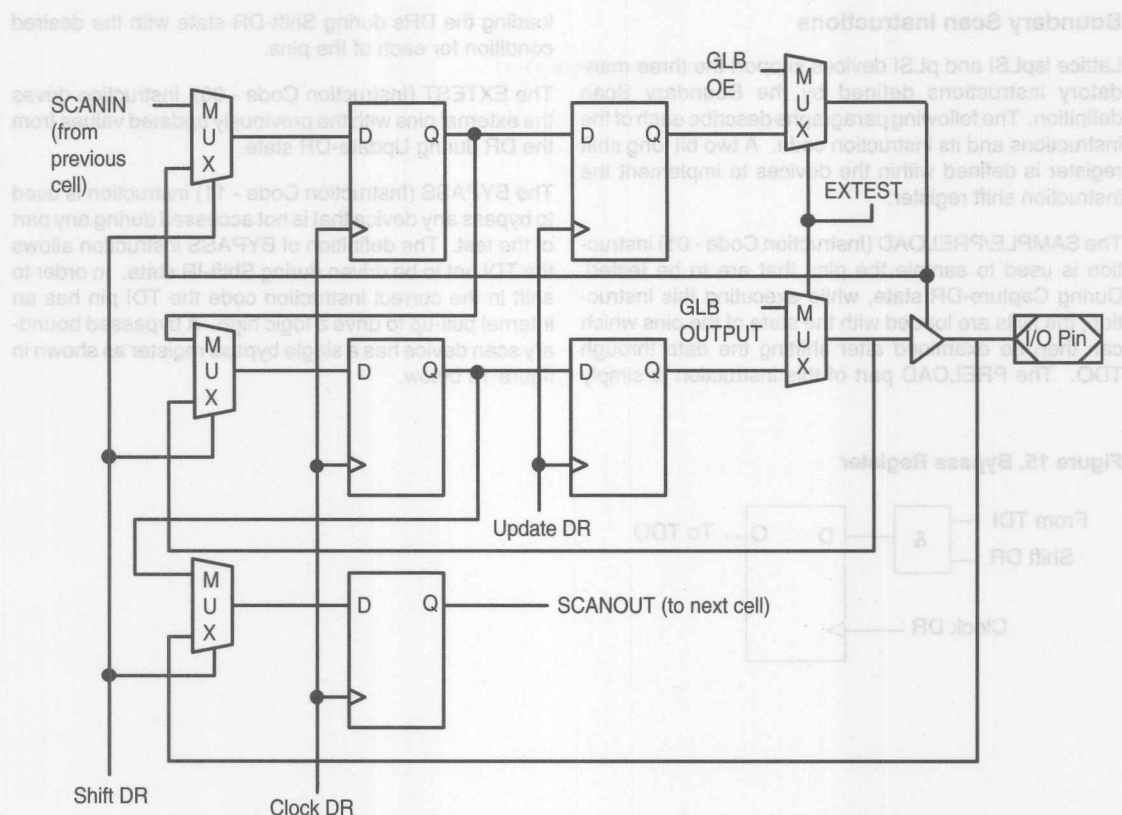


Figure 13. Boundary Scan I/O Cell

and the third captures the input data. These components make up the three registers that are part of the shift register string for each of the I/O pins. Only parts of the I/O cell registers will have valid data when I/O pins are configured as input only or output only and the test routines must be able to monitor the appropriate register bits. The update registers are used mainly to store data that is to be driven onto the I/O pins. The multiplexer controls are driven by the signal from the TAP controller at appropriate states.

The function of an input cell register is much simpler than that of an I/O cell. Figure 14 illustrates the single input register cell. The purpose of the I/O cell is to capture the input test data and shift the data out of the shift register string.

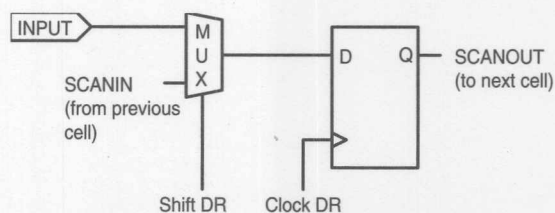


Figure 14. Boundary Scan Input Cell

ispLSI Architecture and Programming

Boundary Scan Instructions

Lattice ispLSI and pLSI devices support the three mandatory instructions defined by the Boundary Scan definition. The following paragraphs describe each of the instructions and its instruction code. A two bit long shift register is defined within the devices to implement the instruction shift register.

The SAMPLE/PRELOAD (Instruction Code - 01) instruction is used to sample the pins that are to be tested. During Capture-DR state, while executing this instruction, the DRs are loaded with the state of the pins which can then be examined after shifting the data through TDO. The PRELOAD part of this instruction is simply

loading the DRs during Shift-DR state with the desired condition for each of the pins.

The EXTEST (Instruction Code - 00) instruction drives the external pins with the previously updated values from the DR during Update-DR state.

The BYPASS (Instruction Code - 11) instruction is used to bypass any device that is not accessed during any part of the test. The definition of BYPASS instruction allows the TDI not to be driven during Shift-IR state. In order to shift in the correct instruction code the TDI pin has an internal pull-up to drive a logic high. A bypassed boundary scan device has a single bypass register as shown in figure 15 below.

Figure 15. Bypass Register

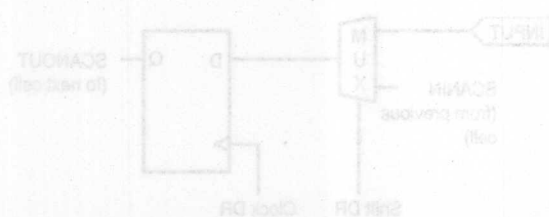
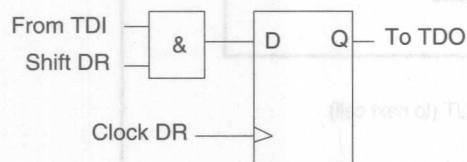


Figure 14. Boundary Scan Input Cell

Figure 13. Boundary Scan I/O Cell

and the third captures the input data. These components make up the three registers that are part of the shift register string for each of the I/O pins. Only parts of the I/O cell registers will have valid data when I/O pins are configured as input or output only and the test routines must be able to monitor the appropriate register bit. The update registers are used mainly to store data that is to be driven onto the I/O pins. The multiplexer controls are driven by the signal from the TAP controller at appropriate states.

The function of an input cell register is much simpler than that of an I/O cell. Figure 14 illustrates the single input register cell. The purpose of the I/O cell is to capture the input test data and shift the data out of the shift register

Section 1: Introduction

Section 2: ispLSI and pLSI Architecture Overview

Section 3: ispLSI and pLSI Development Tools

Lattice Design Tool Strategy	3-1
System Design Process	3-3
ispLSI and pLSI Design Flow	3-5

Section 4: ispLSI and pLSI Application Notes

Section 5: GAL Architecture Overview

Section 6: GAL Development Tools

Section 7: GAL Application Notes

Section 8: In-System Programmable Generic Digital Switch (ispGDS)

Section 9: Design Techniques

Section 10: Article Reprints

Section 11: Technology, Quality, and Reliability Overview

Section 12: General Section

Section 1: Introduction	
Section 2: IqLSI and qLSI Architecture Overview	
Section 3: IqLSI and qLSI Development Tools	
Lattice Design Tool Strategy	3-1
System Design Process	3-3
IqLSI and qLSI Design Flow	3-5
Section 4: IqLSI and qLSI Application Notes	
Section 5: GAL Architecture Overview	
Section 6: GAL Development Tools	
Section 7: GAL Application Notes	
Section 8: In-System Programmable Generic Digital Switch (ISGDS)	
Section 9: Design Techniques	
Section 10: Article Reprints	
Section 11: Technology, Quality, and Reliability Overview	
Section 12: General Section	

Lattice Design Tool Strategy

Introduction

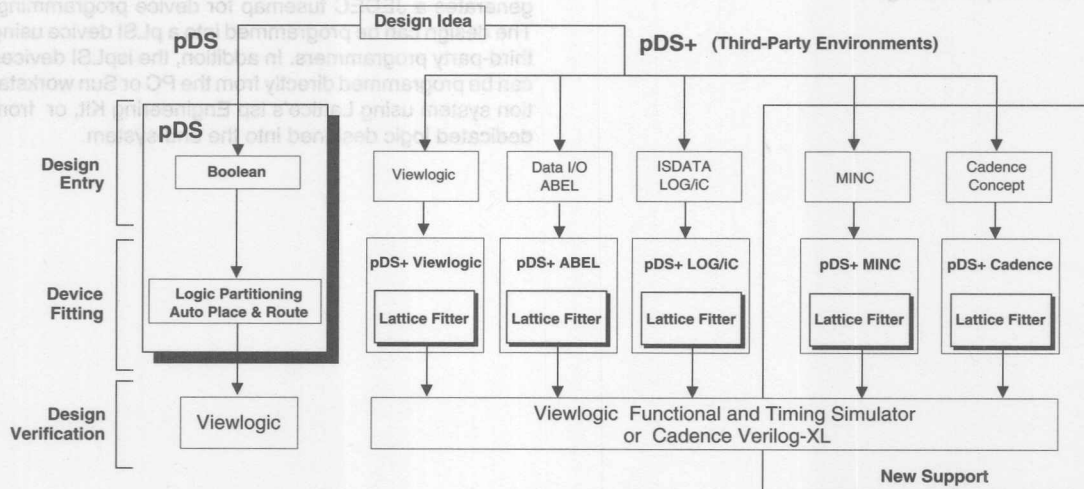
The Lattice design tool strategy for the ispLSI and pLSI families is to support a wide range of design environments. Lattice provides both a proprietary PC-based solution (pDS®) as well as third-party compatible CAE tools (pDS+™ Fitters) that run on PC and Sun workstation platforms.

The Lattice pDS (pLSI and ispLSI Development System) software provides a comprehensive, high-performance, low-cost package for logic development. Developed and supported by Lattice, pDS provides an easy-to-use Windows-based graphical interface using a mouse and pull-down menus. Design entry includes Boolean equations and macros. For simulation, timing tables are included as a standard offering. Additionally, pDS interfaces with Viewlogic's PROsim simulation package for full functional and timing simulation. pDS Software generates industry standard JEDEC programming files and supports direct download into ispLSI devices.

Lattice's pDS+ (pDS Plus) solution supports multiple third-party CAE tools, providing designers with the capability to design in familiar CAE environments. These third-party CAE tools offer schematic capture, hardware description language (such as VHDL), state machine language, Boolean equation, and macro design entry as well as functional and timing simulators for design verification.

Lattice's pDS and pDS+ solutions give designers powerful, easy to use, cost-effective design tools to meet their development needs. Each third-party vendor must adhere to strict quality and certification requirements before becoming qualified, thus ensuring superior support. Contact your local Lattice Sales Representative for availability.

Figure 1. pDS and pDS+ Design Flows



Lattice Design Tool Strategy

Lattice Design Flow

There are three steps in the Lattice ispLSI and pLSI design flow: design entry, device fitting (logic partitioning, place and route), and design verification. (See the pDS and pDS+ Design Flow). This section outlines the design flow of the pDS and pDS+ solutions.

Lattice pDS

Lattice's pDS solution is a comprehensive, self-contained design solution which operates on a PC under Microsoft Windows. pDS uses familiar ABEL-like Boolean equation and macro design entry, and provides manual partitioning, high speed automatic place and route, and simulation timing tables for design verification. Viewlogic's PROsim simulation package is compatible with pDS for functional and timing simulation.

After the development work has been completed, the design is ready to be programmed into a device. For third-party programming support, the pDS package generates a JEDEC fusemap. Alternatively, the ispLSI devices can be programmed directly from the PC or Sun workstation with the Lattice isp Engineering Kit.

The pDS development systems are ideal for designers who desire a cost-effective, user friendly approach to ispLSI and pLSI design.

Lattice pDS+

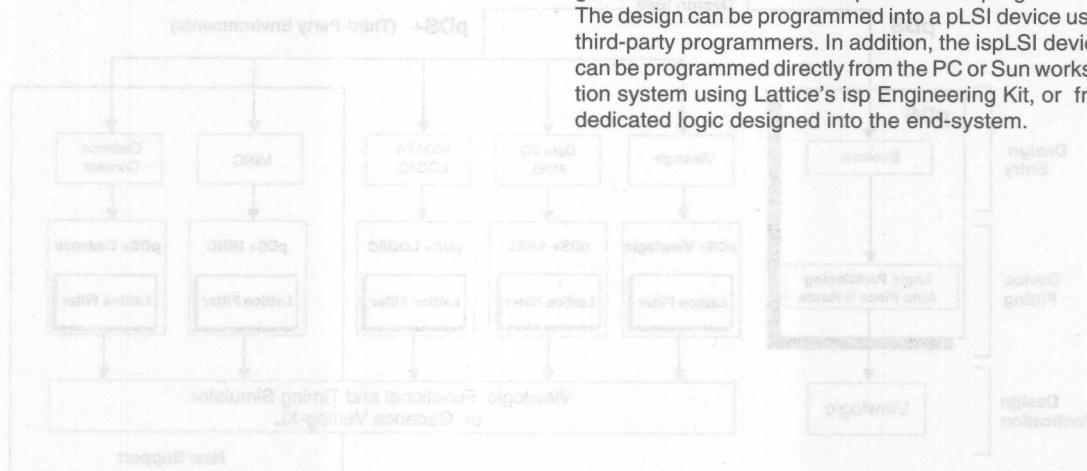
The pDS+ solution combines third-party CAE tools for design entry and verification with the Lattice pDS+ Fitter for device fitting to offer a powerful and complete development solution. Initial Fitter products include the pDS+ ABEL Fitter, pDS+ Viewlogic Fitter, pDS+ LOG/IC Fitter, pDS+ MINC Fitter, and pDS+ Cadence Fitter which interface with their respective third party design tools.

The design entry step is typically performed with schematic capture, Boolean equations, state machines, truth tables or a Hardware Description Language (HDL). Once design entry is complete, the design is ready to be implemented into a Lattice ispLSI or pLSI device.

The Lattice pDS+ Fitter uses architecture-specific algorithms to synthesize a logic description into an ispLSI or pLSI device. Steps in the device fitting process include logic optimization and minimization, automatic logic partitioning, and automatic place and route.

pDS+ also supports design verification. Design verification options include both functional and timing simulation. Various combinations of graphical and text-based functional and timing simulators are supported by third-party CAE vendors.

Following design verification, the Lattice pDS+ Fitter generates a JEDEC fusemap for device programming. The design can be programmed into a pLSI device using third-party programmers. In addition, the ispLSI devices can be programmed directly from the PC or Sun workstation system using Lattice's isp Engineering Kit, or from dedicated logic designed into the end-system.

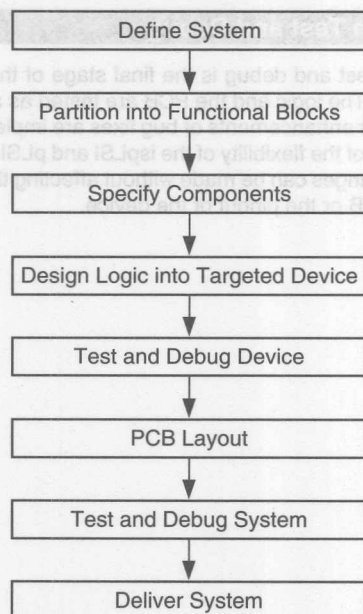


System Design Process

Introduction

Conceptually, system definition is the first step in the design process. This involves visualizing the PLD's interaction with the rest of the electronic system and defining a general flow diagram to determine the design's basic sequential behavior. This organizational flow, used to integrate an entire subsystem into high density devices, is described in the following topics and shown in figure 1.

Figure 1. System Design Flow



Partitioning

After completing the conceptual design, the designer partitions the system into modules or functional blocks. These blocks can be a few components or multiple circuit boards with numerous components. The designer organizes these functional blocks to match the capabilities of the devices being targeted, for example, the number of I/O pins, flip-flops and gates needed. The user should also consider the frequency at which the targeted device must operate, the number of clocks required, and the timing relationships of signals (AC specifications).

Specifying Components

After the partitioning is defined, the designer chooses the components which will be used to implement the desired functions. The design should meet the system specifications using the least number of components in order to keep the system cost as low as possible while keeping the system reliability as high as possible.

System specifications calling for low weight, low power and reduced size also drive designers to higher levels of logic integration. These added requirements can adversely affect the design schedule and project completion. The ispLSI and pLSI high-density devices can meet such design requirements while delivering excellent performance. The ispLSI and pLSI family of high-speed, high-density PLDs supported by easy-to-use effective software for fast design implementation and verification.

Design Entry and Optimization

After the functional partitioning and component specifications are completed, the logic necessary to implement the functions is defined block by block. The logic may include standard TTL functions, CMOS logic functions, or functions from a library, such as the Lattice Macro Library. The implementation of logic into a high density device is optimized for the targeted device by the design software. The partitioning also affects the optimization. Optimization can be for speed, utilization or a combination of both.

Logic entry for a Lattice high density device is done with the pLSI and ispLSI Development System or with any of Lattice's pDS+ Fitter products (pDS+ Viewlogic, pDS+ ABEL, pDS+ LOG/iC, etc.). The pDS Software utilizes the Graphical User Interface (GUI) of Microsoft's Windows™ to provide a complete design flow from logic entry to programming ispLSI and pLSI devices within hours. pDS+ ABEL software supports textual design entry using a Hardware Description Language (HDL) as well as other entry methods. Standard CAE schematic design entry is supported by the pDS+ Viewlogic software. pDS+ LOG/iC supports Boolean, truth table and state machine entry.

System Design Process

Test and Debug

When designing a system, or a portion of a system, it is easier to test and debug pieces or modules rather than the entire system. In this manner, the designer can confirm module designs, or functional blocks, and find problems earlier in the design cycle.

Logic can be verified by either timing simulation or actual testing of the programmed device. Simulation can be accomplished using the Viewlogic Viewsim logic simulator (available from Lattice) and other simulators supported by Lattice. Design errors detected by software simulation can be corrected by the designer before the printed circuit board is laid out and manufactured, which saves time and reduces cost. Board and system level simulation can be accomplished through behavioral simulation using Logic Modeling Corporation's models.

Reprogrammable devices allow the designer to test, debug, and modify logic right on the p.c. board. ispLSI and pLSI devices can be reprogrammed multiple times. This reprogrammability further assists the designers by allowing them to temporarily program the devices with diagnostic and design verification logic.

The designer should always attempt to design logic with testability in mind. Testability means different things to different designers. Key guidelines to be aware of are:

- ☐ Large counters should be segmented for quick and easy testing.
- ☐ Logic should be designed for controllability and observability.
- ☐ There should be no floating nets.
- ☐ All nets should be at a known state or are able to be set or reset.

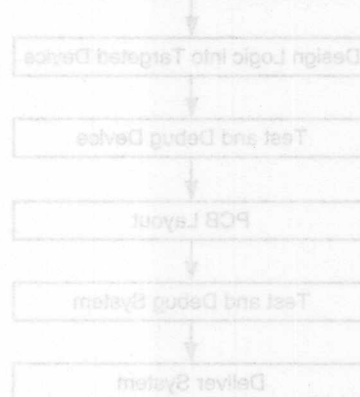
To assist system testability, the ispLSI devices offer preload and verification features. These features allow register contents to be verified without using logic analyzers or other debugging tools.

Printed Circuit Board Layout

Once the logic has been verified, the Printed Circuit Board (PCB) is laid out and manufactured. Since the logic may be changed during design, this phase of the system design is usually executed after the logic has been validated. It is recommended that board design and layout be done after verifying designs using ispLSI and pLSI parts.

System Test and Debug

System test and debug is the final stage of the design process. The logic and the PCB are tested as a system and minor enhancements or bug fixes are implemented. Because of the flexibility of the ispLSI and pLSI devices, minor changes can be made without affecting the layout of the PCB or the pinout of the device.



After completing the conceptual design, the designer partitions the system into modules or functional blocks. These blocks can be a few components or multiple circuit boards with numerous components. The designer organizes these functional blocks to match the capabilities of the devices being targeted, for example, the number of I/O pins, flip-flops and gates needed. The user should also consider the frequency at which the targeted device must operate, the number of clocks required, and the timing relationships of signals (AC specifications).

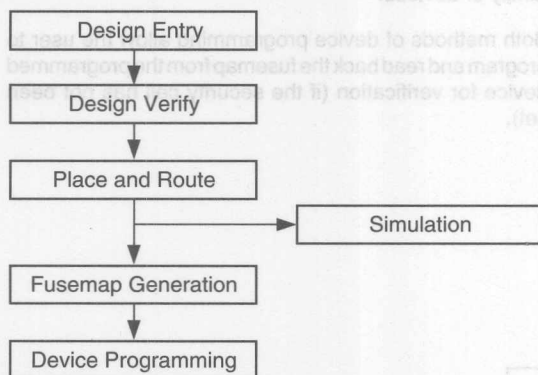
ispLSI and pLSI Design Flow

3

Introduction

Once the system design has been organized into functional components, and the logic functions which need to be incorporated in the selected components defined, the logic design phase begins. The general design flow is shown in figure 1. An ispLSI or pLSI design may be implemented from a number of design environments. This section will discuss four popular ones: pLSI and ispLSI Development System (pDS), pDS+ ABEL, pDS+ Viewlogic, pDS+ LOG/iC, pDS+ MINC, and pDS+ Cadence.

Figure 1. General Design Flow



These design environments offer various levels of design implementation from logic entry through programming the device. They support a variety of user interfaces and entry methods including: MS Windows GUI, Data I/O ABEL HDL or VHDL, Viewlogic Viewdraw/Viewsynthesis and PROcapture/PROsynthesis, ISDATA LOG/iC Design System, Cadence Concept/Verilog-XL, and MINC PLDesigner-XL. The design flows using these development software systems are shown in figures 2, 2a, 2b, 2c, 2d and 2e.

Design Entry

The pDS Software allows the user to manually partition the logic to control design fit and performance. Using the MS Windows environment, logic functions are placed into Generic Logic Blocks (GLBs) and I/O Cells. This can be done by using the Edit, Cut, Copy, and Paste functions to enter Boolean equations and/or predefined functions from the Lattice Macro or user libraries.

In addition to Boolean design entry, the ABEL HDL and MINC HDL and VHDL formats allow high-level descriptions of counters, adders, comparators, etc. These HDL languages also support state machines, truth tables and case constructs for behavioral design implementations. The Lattice interfaces allow many existing PLD designs to be easily integrated and converted into an ispLSI or pLSI devices.

For standard CAE schematic designs, the pDS+ Viewlogic and pDS+ Cadence software provide support for graphical and hierarchical logic implementations using the Lattice library of primitives and macros. The interfaces also allow easy integration of system or user-created functions into a hierarchical schematic using a top-down or bottom-up design methodology.

Design Verification

Verification using the pDS Software is accomplished in two steps after logic has been placed. First, each cell may be individually verified to ensure that the minimized logic will fit into the GLB architecture. After all GLB and I/O cells are incrementally checked, the entire design is then verified to ensure that all nets have proper sources and destinations.

Because the advanced pDS+ tools perform automatic partitioning, design verification is done at a higher-level (pre-partitioned). For example, in the ABEL environment the Compile (ahdl2pla) function performs the syntax and design rule checks. After the Compile phase, the Optimize (plaopt) function (optionally) minimizes the design.

In other pDS+ environments, pre-partitioned design verification is performed by the Design Analyzer which ensures the logic conforms to the Lattice design rules.

Partitioning

Partitioning using the pDS Software is done by the user as part of the design entry process. The advanced pDS+ Fitter tools incorporate Lattice's automatic partitioner which accepts converted data from designs entered in ABEL, Viewlogic, LOG/iC, MINC and Cadence tools. Lattice specific attributes for design entry are available to guide the partitioner in order to optimize usage of device features and performance.

ispLSI and pLSI Design Flow

Place and Route

All Lattice design tools offer automatic place and route. This entails placement of GLB and IOC logic and then routing (or interconnecting) the source signals to their destinations. In the ispLSI and pLSI devices, the Global Routing Pool (GRP) provides fast interconnects from external inputs and GLB feedbacks to the GLB inputs. The Output Routing Pool (ORP) provides flexible interconnects from GLB outputs to external pins. To take advantage of the architectural features, Lattice offers an extended route option for more comprehensive routing of complex designs.

Post-route Simulation

After place and route, a netlist for full timing and function simulation may be passed to the Viewsim or Verilog-XL simulator. Viewsim supports simulation using both textual and graphical input and interfaces. Board and system level simulation models are also available from Logic Modeling Corporation.

Documentation

Report files, containing partitioned equations and pin-out information, may be generated for routed or un-routed designs. The pDS Software can also generate reports with post-route maximum timing delays.

Device Programming

Programming information is generated on a routed design by the FuseMap Generator for a specific ispLSI and pLSI device. It is an ASCII file written in the JEDEC format. Using ABEL and MINC software, the user may optionally append test vectors onto the JEDEC file. This allows post-programming functional test on the actual device.

Two programming methods are used to program the ispLSI and pLSI devices. The first method uses the Device Programming Mode for both types of devices. This method facilitates device programming support from third-party vendors. The second method uses the Lattice In-System Programming Mode and applies to the ispLSI family of devices.

Both methods of device programming allow the user to program and read back the fusemap from the programmed device for verification (if the security cell has not been set).

Figure 2. pDS Design Flow

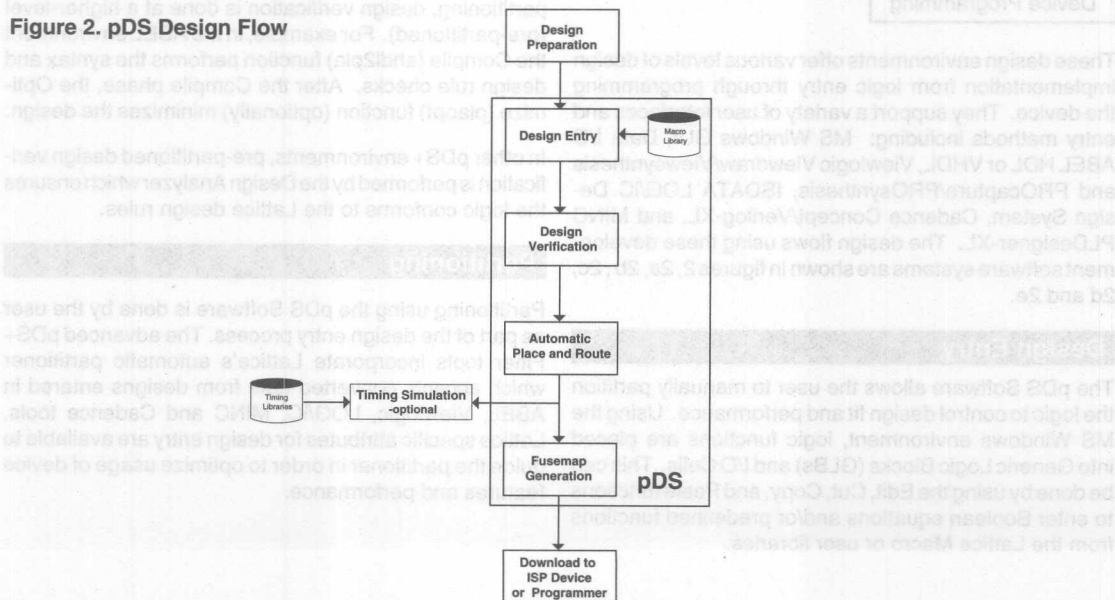


Figure 2a. pDS+ Viewlogic Design Flow

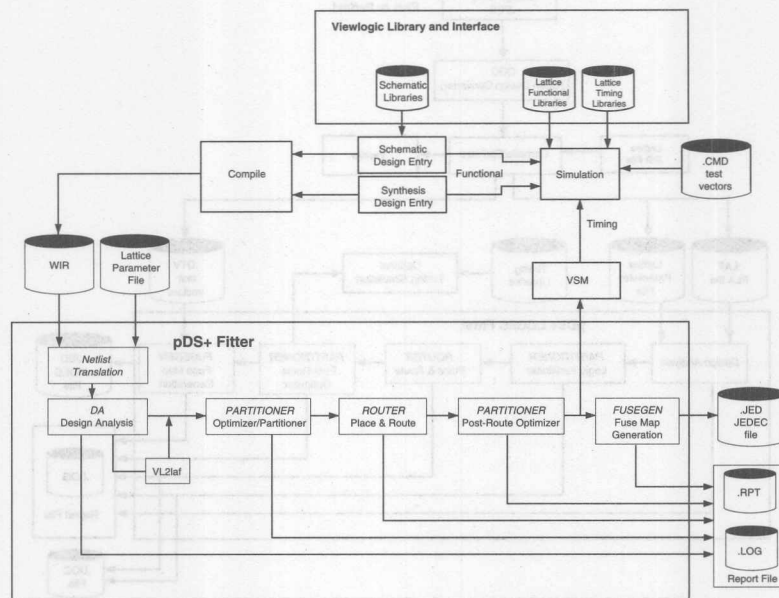


Figure 2b. pDS+ ABEL Design Flow

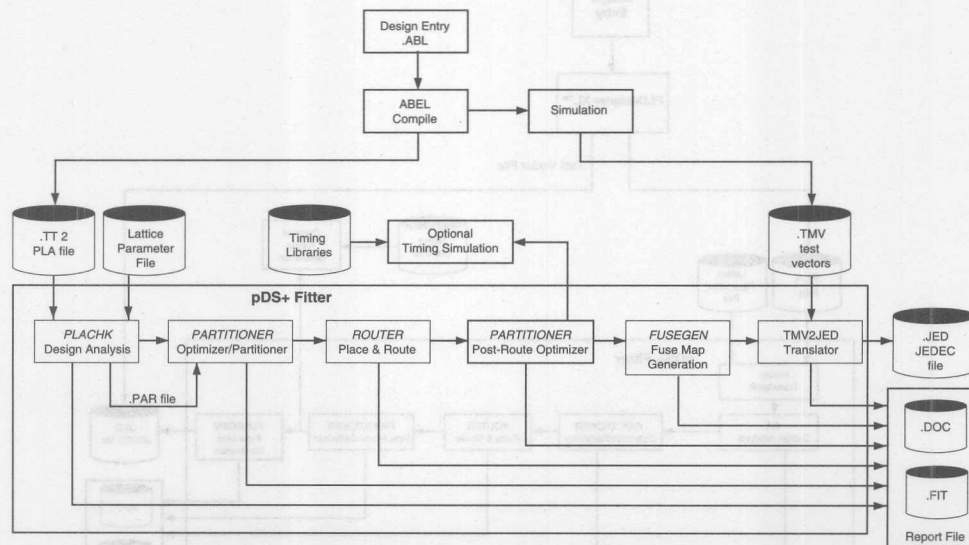


Figure 2c. pDS+ LOG/IC Design Flow

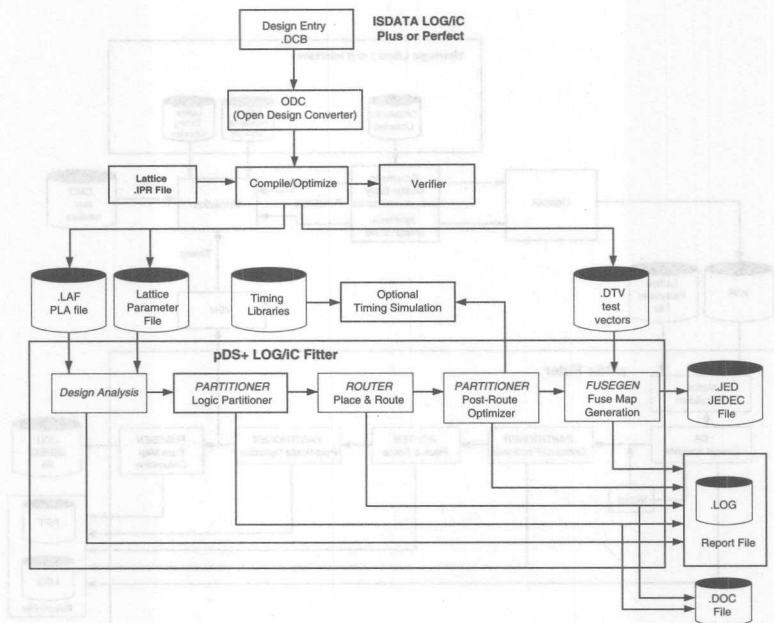


Figure 2d. pDS+ MINC Design Flow

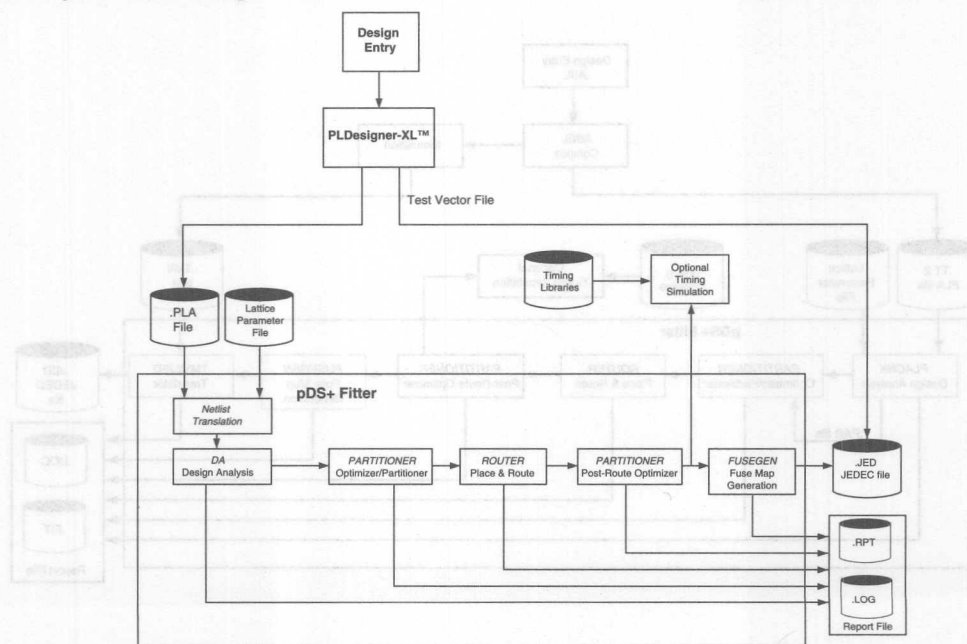
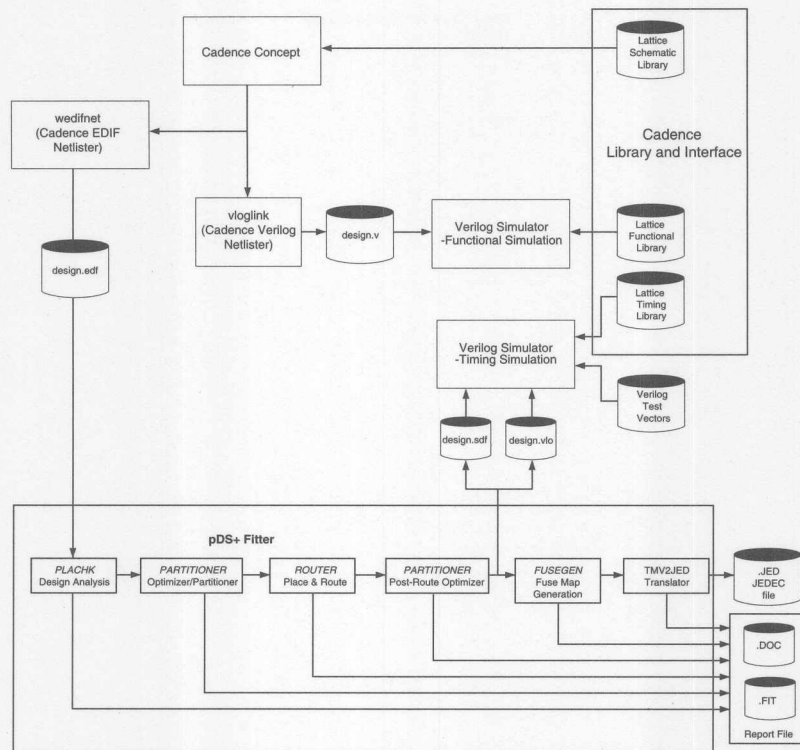
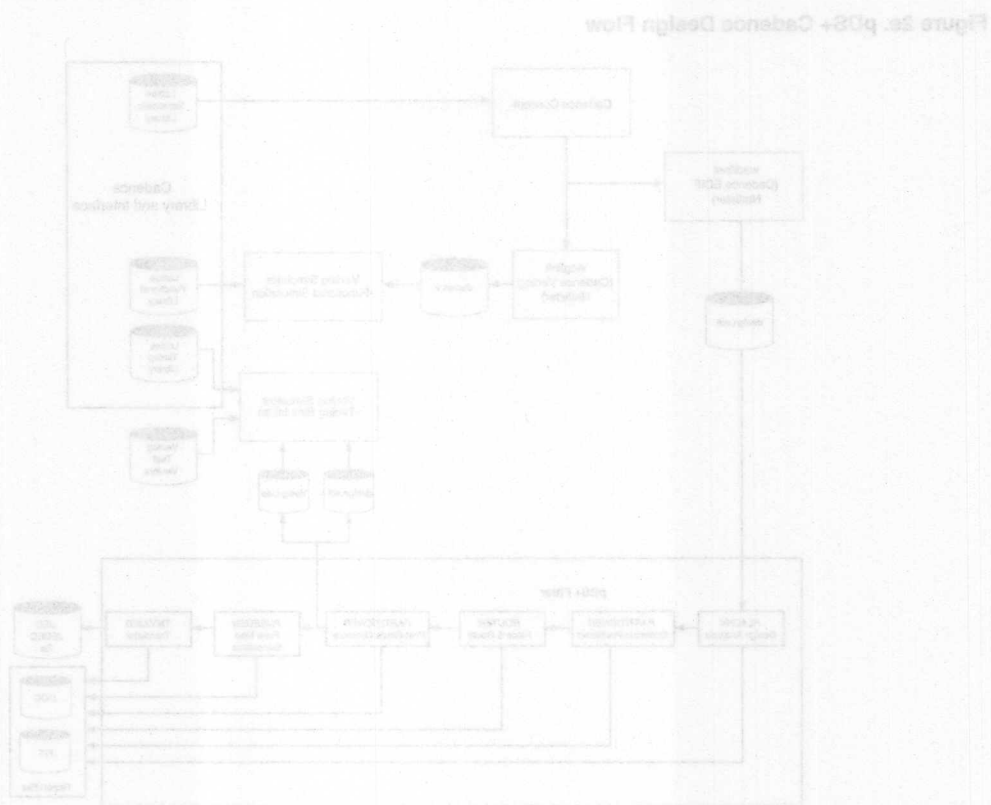


Figure 2e. pDS+ Cadence Design Flow



Notes



Section 1: Introduction

Section 2: ispLSI and pLSI Architecture Overview

Section 3: ispLSI and pLSI Development Tools

Section 4: ispLSI and pLSI Application Notes

Selecting the Right High Density Device	4-1
Beginner's Guide to ispLSI and pLSI	4-7
ispLSI and pLSI: A Multiple Function Solution	4-21
Programming Multiple ISP Devices: Daisy Chain Configuration	4-41
Compiling Multiple PLDs into ispLSI and pLSI Devices	4-47
Adders/Subtractors in pLSI	4-55
Crosspoint Switch Implementation Using the pLSI 1032	4-61
Building Modulo N Counters Using ispLSI and pLSI Devices	4-69
Phase Locked Loops (PLL) in High Speed Designs	4-71
Video Graphics Controller	4-75
A Digital Clock Design Example	4-95
ispLSI Configurable Memory Controller	4-105
Bar Code Reader	4-121
High Density PLD Solutions for High Speed RISC/CISC Systems	4-139
SCSI Interface with the ispLSI 3256	4-145
PCI Bus Implementation	4-155
Programming ispLSI Devices with a Tester	4-179

Section 5: GAL Architecture Overview

Section 6: GAL Development Tools

Section 7: GAL Application Notes

Section 8: In-System Programmable Generic Digital Switch (ispGDS)

Section 9: Design Techniques

Section 10: Article Reprints

Section 11: Technology, Quality, and Reliability Overview

Section 12: General Section

Section 1: Introduction

Section 2: iaplsi and plsi Architecture Overview

Section 3: iaplsi and plsi Development Tools

Section 4: iaplsi and plsi Application Notes

4-1	Selecting the Right High Density Device
4-5	Beginner's Guide to iaplsi and plsi
4-21	iaplsi and plsi: A Multiple Function Solution
4-41	Programming Multiple ISP Devices: Daisy Chain Configuration
4-47	Compiling Multiple PLDs into iaplsi and plsi Devices
4-55	Address Subtraction in plsi
4-57	Crosspoint Switch Implementation Using the plsi 1035
4-59	Building Module N Counter Using iaplsi and plsi Devices
4-71	Phase Locked Loops (PLL) in High Speed Designs
4-75	Video Graphics Controller
4-95	A Digital Clock Design Example
4-105	iaplsi Configurable Memory Controller
4-121	Bar Code Reader
4-139	High Density PLD Solutions for High Speed RISC/CISC Systems
4-145	SCSI Interface with the iaplsi 3255
4-155	PCI Bus Implementation
4-175	Programming iaplsi Devices with a Tester

Section 5: GAL Architecture Overview

Section 6: GAL Development Tools

Section 7: GAL Application Notes

Section 8: In-System Programmable Generic Digital Switch (ispGDS)

Section 9: Design Techniques

Section 10: Article Reprints

Section 11: Technology, Quality, and Reliability Overview

Section 12: General Section

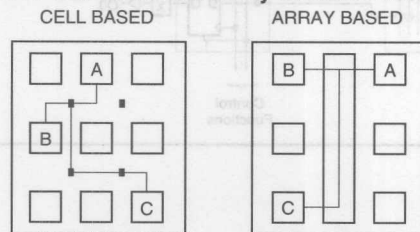
Introduction

Board designers today have several options for implementing their designs in high density programmable devices. Due to technology and design considerations, no single device provides the best solution for the challenges facing designers. To address this, design engineers often use multiple types of high density devices on a single board. This paper will outline various applications issues and examine the appropriate high density solutions. It will also examine from the perspective of the user, the impact of design implementation, on the process of selecting a device.

High density programmable devices can be broadly classified into two major types: Field Programmable Gate Arrays (FPGA) and High Density Programmable Logic Devices (HDPLD). FPGA devices are cell based and usually have small grain-size logic blocks with distributed interconnects across the device. High Density Programmable Logic Devices are array based and have large grained AND-OR array logic blocks with centralized interconnects (see figure 1). Similarly, board designs can be broadly classified into two types: control intensive and data intensive. Control intensive designs, usually contain such subfunctions as Cache control, DRAM control, DMA control and require limited data manipulation. Data intensive designs, on the other hand, require complex manipulation of data bits which are typically found in telecommunications type applications. To select a high density device a designer must examine:

- ☐ Performance
- ☐ Utilization
- ☐ Ease of Use

Figure 1. Cell based and Array based Devices



Performance

When implementing a logic design into a high density device, it is typically partitioned into multiple logic blocks or cells and then the various cells are connected together using interconnect resources. The performance of a design is determined by the combination of the cell speed and the interconnect speed.

Cell Speed

A logic function is divided into subfunctions which fit the basic building block of the high density device. Often the number of inputs is the most important consideration. The subfunctions should require no more inputs than are available in the logic block of the device. Smaller logic blocks tend to be faster but they offer fewer inputs. Functional implementation often requires a number of logic blocks cascaded into multiple levels of delay to implement the logic. This slows down the functional speed of the logic dramatically. Control functions are typically input intensive and will be faster in devices with building blocks that allow for a large number of inputs. Data functions require fewer inputs and may be faster in devices which have fewer inputs per logic block.

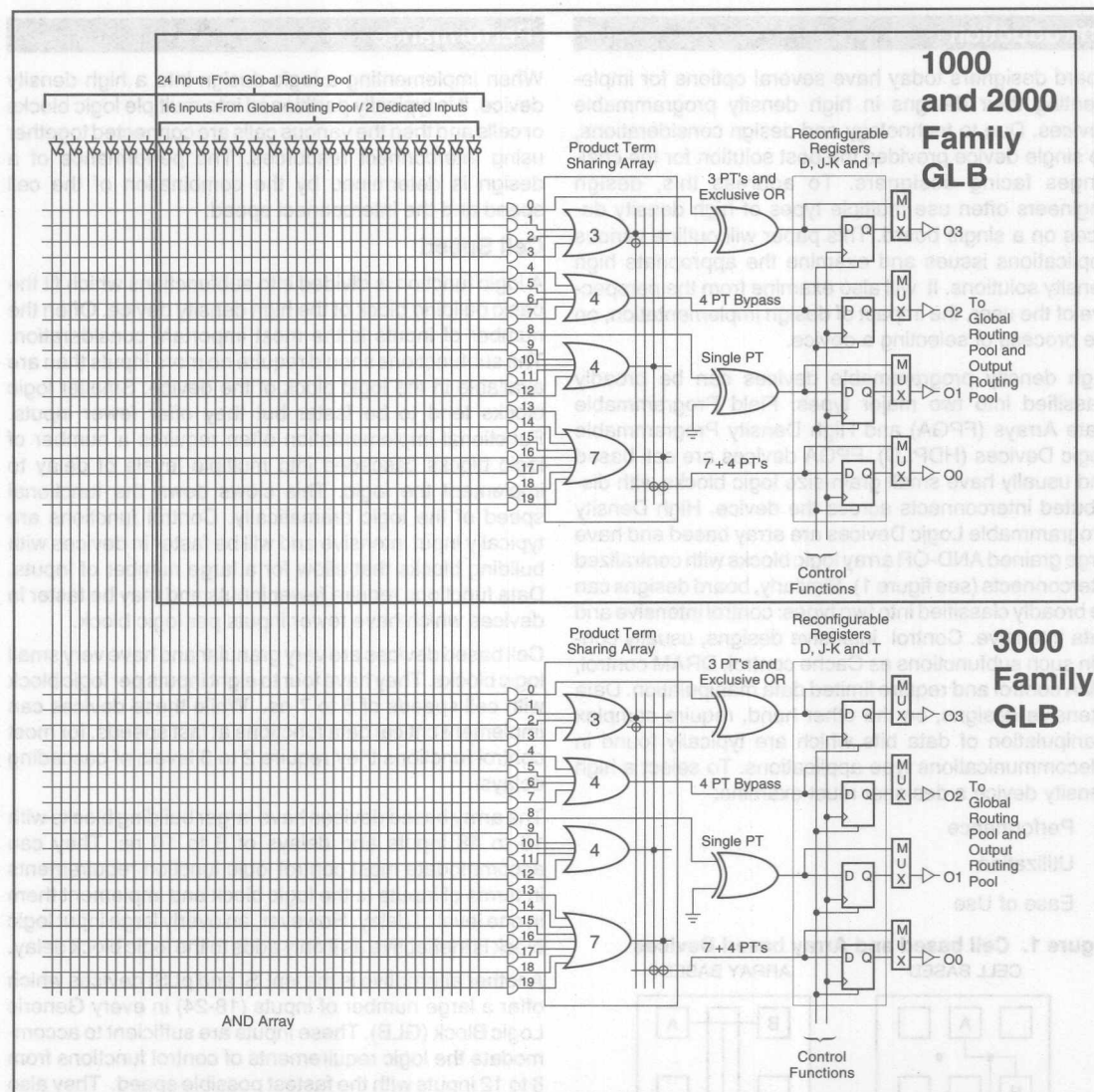
Cell based devices are very granular and have very small logic blocks. They have four to eight inputs per logic block with cell speeds of 6 to 7 ns. While these devices can implement critical data functions at fast speeds, for most control functions they require 2 to 3 levels of cascading delays.

The array based devices have larger building blocks with 16 to 48 inputs and delays of 8 to 10 ns. They can accommodate most control logic function requirements in terms of inputs to the logic block and implement them in one level of delay. However, an overly large input logic block is ineffective as it only adds to the logic block delay.

Another alternative is the ispLSI and pLSI devices which offer a large number of inputs (18-24) in every Generic Logic Block (GLB). These inputs are sufficient to accommodate the logic requirements of control functions from 8 to 12 inputs with the fastest possible speed. They also accommodate data functions which require 2 to 4 inputs per output, while maintaining high speed.

Selecting the Right High Density Device

Figure 2. pLSI GLB



0845

Selecting the Right High Density Device

The graph illustrates (see figure 3) multiple logic block delays required to perform common logic functions in some of the popular high density devices available today. Due to a limited number of inputs available in Cell based devices the number of logic blocks cascaded to perform a function can be as high as 4 to 7. The Array based devices (vendor C) require only one level of delay. ispLSI and pLSI devices require only one logic block for most functions. The logic block delay is small and is comparable to most Cell based devices.

Interconnect Speed

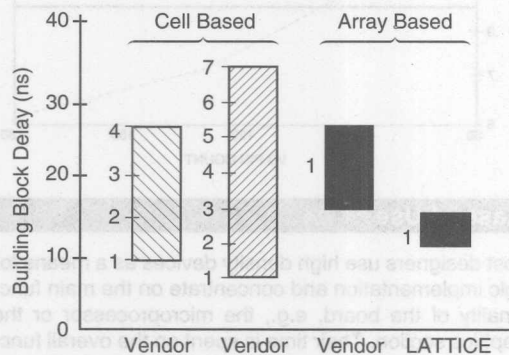
Interconnect speed is another important consideration not only for connecting subfunction logic blocks, but also for connecting signals from one logic function to another. Interconnects affect final system performance as much as logic blocks do.

The cell based devices offer distributed interconnects with variable length lines spanning the length and the width of the device and interconnecting various logic blocks with finite delay interconnect points. Frequently, signals have to traverse multiple line segments and

placed in close proximity. Frequently, this is physically impossible and/or requires many placement iterations. Often, for such designs as state machines, counters, etc., the final performance is determined by the worst case signal speed. In such cases, the cell based devices with distributed interconnects offer slower interconnect performance and consequently slower overall system performance. Data functions require fewer delays and can be implemented relatively faster. Placement of related data bits close to each other facilitates fast performance for data oriented functions. However, sometimes with a large number of data bits this becomes difficult to achieve.

The array based devices with centralized interconnects offer uniform interconnect delays. The ispLSI and pLSI devices offer uniform interconnect delays with significantly faster interconnect performance than existing array based devices, and consistently provide best case cell based device delays as illustrated in Figure 4.

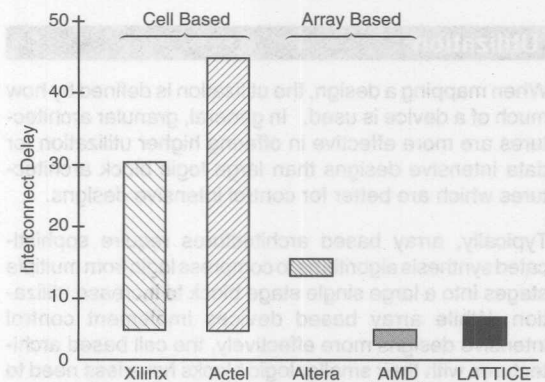
Figure 3. Building Block Performance



SOURCE: Lattice Applications, Vendor Literature

interconnect points for a connection. In general, closely located logic blocks have a shorter delay since signals travel through fewer lines and interconnect points. The opposite is true for logic blocks located further apart. There is a large variation in the interconnect delays based on the placement of the related logic blocks (see figure 4). In general, to improve system performance for control oriented functions in a cell based device, a large number of signals and related logic blocks need to be

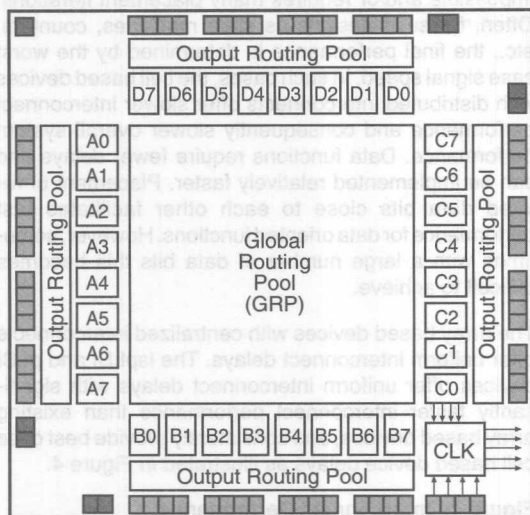
Figure 4. Interconnect Performance



The state-of-the-art for Automatic Place and Route (APR) software has not reached a point where the interconnect performance can be called optimal. In general, the cell based devices either require a long APR time, typically a number of hours or days, in order to reach near-optimal interconnect speeds. Shorter route times result in reasonable performance. Uniform delays in array based devices eliminate the need for intelligently placing related logic blocks closer, thereby reducing APR time to a few minutes. ispLSI and pLSI devices go a step further and offer faster interconnect delays using the proprietary centralized Global Routing Pool (GRP) (see figure 5) which retains fast APR times. This is especially good for data intensive designs where all data bits perform equally.

Selecting the Right High Density Device

Figure 5. pLSI 1032 Block Diagram



Utilization

When mapping a design, the utilization is defined by how much of a device is used. In general, granular architectures are more effective in offering higher utilization for data intensive designs than large logic block architectures which are better for control intensive designs.

Typically, array based architectures require sophisticated synthesis algorithms to compress logic from multiple stages into a large single stage block to increase utilization. While array based devices implement control intensive designs more effectively, the cell based architectures with their smaller logic blocks have less need to compress logic into one stage of the design. Data intensive designs which typically require a large number of registers are more effectively implemented in a cell based architecture which have higher register to logic ratios.

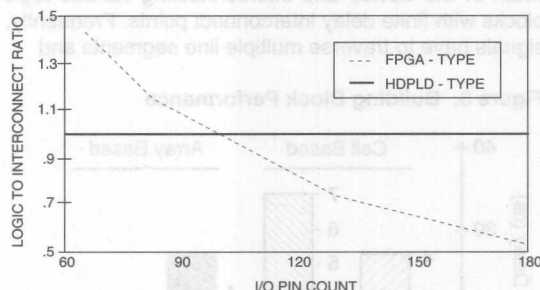
The ispLSI and pLSI devices are neither as granular as some cell based architectures, nor as large as some array based architectures, since they offer a grain size of four outputs per logic block (GLB). However, they offer effective utilization because real life designs are a combination of data and control functions.

Within each GLB, ispLSI and pLSI devices offer a Product Term Sharing Array (PTSA). The PTSA optionally shares GLB product terms between the four GLB outputs thereby enhancing logic block utilization.

As devices are scaled to higher densities, the interconnect resources should increase at the same pace as the logic resources. This ensures that all of the available logic is fully utilized in the device. The distributed nature of cell based interconnects does not lend itself well to this scaling. Figure 6 shows the logic-to-interconnect ratio for one of the families of cell based high density devices. At higher densities, this means a lower utilization of the device since the logic cannot be mapped as easily as at the lower end of the spectrum. The array based devices scale the interconnect resources at the same level as the logic resources and offer better utilization at the higher end range of devices as well.

The ispLSI and pLSI Global Routing Pool (GRP) provides all signals globally to all device GLBs. The GRP size is scaled to provide full 1:1 logic to interconnect ratio ensuring all device logic is fully utilized, irrespective of the device size.

Figure 6. Logic to Interconnect Ratio



Ease of Use

Most designers use high density devices as a means for logic implementation and concentrate on the main functionality of the board, e.g., the microprocessor or the graphics section. Their time is spent on the overall functionality of the board and not on the basic logic. Ease of use and quick design turnaround times are critical to any digital designer. Ease of use is determined by a number of factors. Some critical factors directly related to the choice of device architecture are:

- ☐ Predictability of Performance
- ☐ Design Rework
- ☐ Design Entry
- ☐ Turnaround time

Selecting the Right High Density Device

Predictability of Performance

The performance of the design is determined by the system considerations and is usually driven by the processor requirements or other considerations like graphics screen resolution, etc. High density devices frequently do not determine the final system speed. Designers will need to know in advance the final performance of the logic implemented in the high density device to determine the feasibility of the part selected. A designer also needs to know the speed grade required in advance, in order to estimate the cost of the design.

For cell based devices, the number of delay levels it would take to implement the design function is not typically known. Also modifications to the design often may cause a change in the number of delay levels. Similarly, it is difficult to predict how the software will place the logic blocks as explained earlier. Even if only one out of ten critical signals ends up being slow, it will slow down the device system speed. Array based devices as well as ispLSI and pLSI devices, have predictable levels of logic and interconnect delays, which allows the designer to not only estimate speed in advance but maintain fast speeds.

Design Rework

Very few designs work the first time after entering the logic into a device. Most designs not only require logic addition or subtraction but frequently require pinout changes, and rework. This rework is often due to logic debugging and can sometimes be due to changes in the specification of the final product, etc. Also a large category of digital designers prefer an incremental design approach where small chunks of designs are implemented at a time and debugged before new chunks are added.

For cell based devices every logic change requires a new set of logic mappings into the device cells and a new set of interconnect mapping into device interconnect lines. This leads to significant changes in the performance of the device and to undesired pinout change.

Array based devices typically do not have any adverse performance changes due to logic changes. However, pinout changes frequently occur.

ispLSI and pLSI devices were developed to allow users to make logic changes without any performance impact and to freeze pinouts when incremental design changes

are done. ispLSI and pLSI devices offer an Output Routing Pool (ORP) which allows GLB outputs to be routed to many different I/O pins. Also the ispLSI and pLSI GRP allows I/O pin inputs to be available to all GLBs. These two features when combined offer the flexibility necessary to maintain pinouts in subsequent iterations of designs while maintaining the same performance.

Design Entry

There are two categories of digital designers using high density devices. The first is the designer who is a PLD user, such as with GAL devices and is familiar with Boolean, State Machine or HDL type of design entry syntax. For these designers, array based devices offer direct mapping correlation from the entry syntax to design implementation, which is very helpful in control intensive designs. This makes such factors as the logic implementation, speed of functions and race conditions etc., predictable to the designer and simplifies the design task. The other category is the gate array designer who migrates to programmable gate array devices. For these designers the cell based devices offer a closer correlation between schematic entry to actual design implementation. These designers also implement data intensive designs effectively, since they have a number of TTL type data function macros available to them. With synthesis techniques however, the schematic entry is also offered for array based devices. Familiar design entry methodology also speeds design entry time and simplifies the design process.

ispLSI and pLSI devices offer direct correlation with Boolean/HDL/State Machine entry syntax. Extensive synthesis techniques are also used in the ispLSI and pLSI Development System software to offer easy schematic capture along with a large library of TTL-type Macros.

Turnaround Time

Once a design is entered, the next critical step is design compilation and programming of the part. For cell based devices with smaller logic blocks, the distributed nature of interconnects complicates matters. The Place and Route algorithm needs to satisfy multiple and often conflicting requirements for:

- ☐ Placing Related Logic Closer for Faster Speed.
- ☐ Moving Logic to Satisfy Critical Timing Requirements.

Selecting the Right High Density Device

- ❑ Moving Logic Due to the Lack of Interconnect Resources.
- ❑ Repartitioning Logic to Satisfy the Above Three Conditions.

These four basic requirements are interrelated and complex, making the compilation process very time consuming. Typical cell based devices require 2 to 8 hours for compilation in order to achieve reasonable system performance objectives.

The array based devices with global connectivity and uniform interconnect delays eliminate the need to closely place related logic. The ispLSI and pLSI devices with centralized interconnect offer compilation times of minutes versus hours for the cell based devices thereby improving designer productivity. This combined with the ispLSI version of the family which allows on-board reprogramming of multiple devices simultaneously, offers a whole new dimension for logic design.

Conclusion

A digital designer has multiple choices available for high density designs. The current solutions broadly categorized as cell based and array based devices offer alternative advantages and disadvantages for digital designers. The type of solution chosen depends upon the type of logic functions being implemented, the performance required and other design specific trade-offs. The ispLSI and pLSI devices offer the advantages of both cell based and array based devices.

High Performance - ispLSI and pLSI devices are designed to be extremely fast for both control and data intensive functions and are particularly excellent for functions requiring more than eight inputs per logic block.

Predictable Delays - The centralized GRP structure combined with wide input GLBs offers uniform delays which allows the designer to determine system speed in advance as well as maintain constant speeds in subsequent iterations of the design.

High Utilization - The ability to scale interconnect resources at the same level as logic resources combined with built-in flexibility of the GRP and ORP assure high device utilization. Also the PTSA adds another level of flexibility for increasing logic block utilization.

Ease of Use - Predictable performance, quick design entry and rework time provide fast design turnaround. This simplifies the design process and enhances time to market.

The combination of high performance, predictable delays, high utilization and ease of use, not only offers a superior solution for design requirements; it is delivered in E²CMOS technology with reprogrammability and 100% testability offering unparalleled device quality.



Beginner's Guide to ispLSI and pLSI

Introduction

This Beginner's guide is designed to help you become familiar with the Lattice pLSI 1032 device, ispLSI 1032 device and the Lattice pLSI and ispLSI Development System (pDS™). To do this, a complete design of a simple four-bit counter is discussed from specification through programming and testing the part. The following assumptions are being made. First, you have read and understood the pLSI 1032 data sheet. Next, you have the documentation for Microsoft® Windows™ readily available. Everything else should be here in this Beginner's Guide.

The Lattice pDS Software is designed to run under Microsoft Windows Version 3.1 or later (see figure 1).

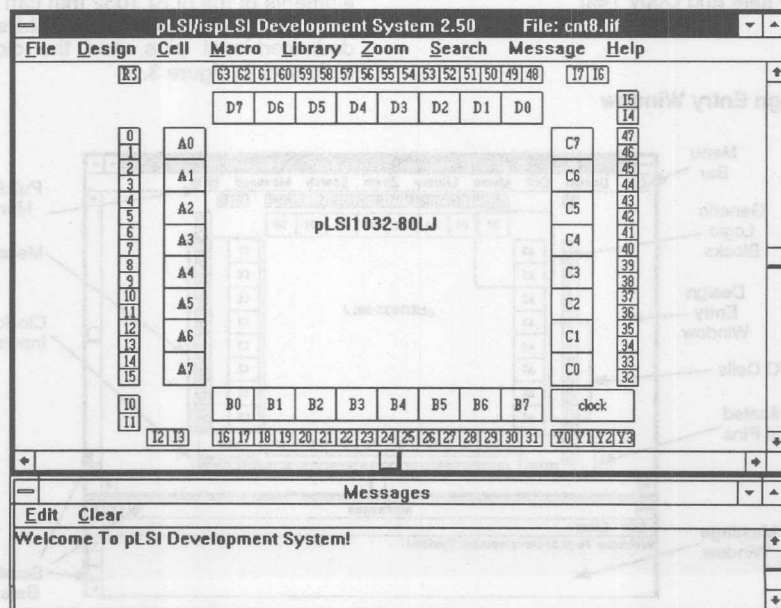
Windows is an industry standard Graphical User Interface (GUI) for pull down menus, text editing commands and screen control commands. Because the Lattice interface is the same as other Windows programs, it is very easy to learn. If you know how to run any Windows program, you can run the Lattice software.

It is necessary to have Windows for the Lattice pDS Software to run. Windows runs on most standard IBM PCs or clones. If your computer runs Windows 3.1, it will run the Lattice pDS Software. The recommended system configuration for running pDS Software is:

- ☐ A 386 or 486 Processor
- ☐ 4 Megabytes of RAM
- ☐ 40 Megabyte Hard Disk
- ☐ A Floppy Disk Drive
- ☐ A Microsoft Windows Compatible Mouse
- ☐ VGA or Super VGA Graphics

In addition, the pDS Software requires that either a spare parallel printer port be available to perform in-system programming, or a spare serial port be available to communicate with an RS-232 controlled programmer.

Figure 1. Lattice pDS Software Opening Screen



Beginner's Guide

Getting Started

If you have not previously installed the Lattice pDS Software, see the installation procedure which came with your development system.

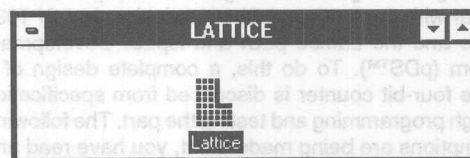
1. To start Windows, type WIN at the DOS Prompt (C: >).
2. Install the Lattice pDS Software according to the installation instructions. A new program group called LATTICE is created. This program group should contain a single icon, called LATTICE.EXE, which looks like the Lattice company logo (see figure 2).
3. To start the pDS Software, double click on the Lattice Logo Icon.

Before you can proceed any further, some of the Microsoft Windows tasks that you should be able to perform are:

- ☐ Selecting a Menu Item Using the Mouse
- ☐ Using Open, Save and Save As Menu Items
- ☐ Entering Commands and Text into Message Windows and Dialog Boxes
- ☐ Moving Around the Screen with the Scroll Bars
- ☐ Editing Text Using the Keyboard and Mouse to:
 - Select the Insertion Point
 - Select Text by Highlighting It
 - Cut, Paste and Copy Text

If you are unfamiliar with any of these options, then take some time to go through the Windows Users Guide. If you have ever worked with the Apple™ Macintosh™, you will find that many of the commands and operations are similar.

Figure 2. Lattice Program Group Window



A Brief Tour of the Screen

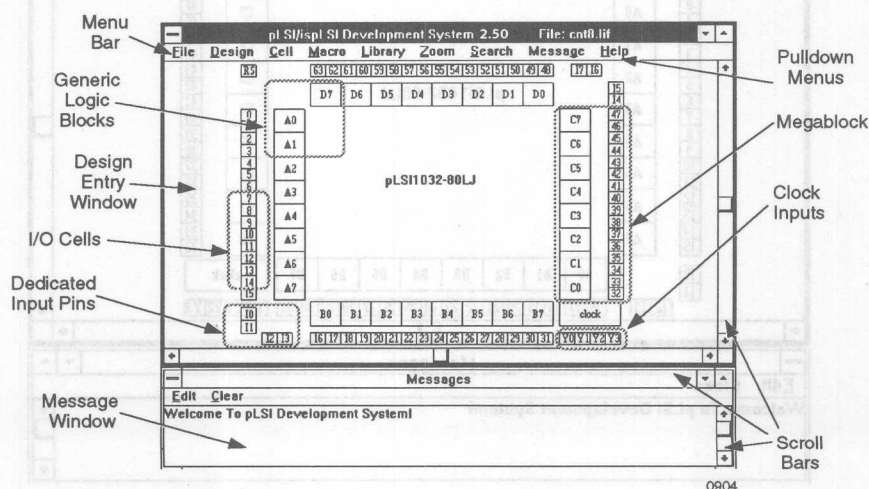
Once you invoke the Lattice pDS Software, two windows are displayed (see figure 8-9).

The larger of the two windows displays a graphical representation of the pLSI 1032 logic diagram. This window is called the Data Entry Window. The design is entered by editing equations in the Data Entry Window.

The smaller of the two windows is the Message Window and it is located at the bottom of the screen. The pDS Software communicates with you by placing messages in the message window.

The part that is displayed in the block diagram shows the elements of the pLSI 1032 that can be modified by the user. These elements are the GLBs, the I/O Cells, the dedicated input pins, and the clock input pins, as indicated in figure 3.

Figure 3. Design Entry Window



The design is entered into the development software by clicking on the block that you wish to edit and entering equations or Macros (library elements already partitioned and optimized for high performance) into the Edit Window that appears (see figure 4).

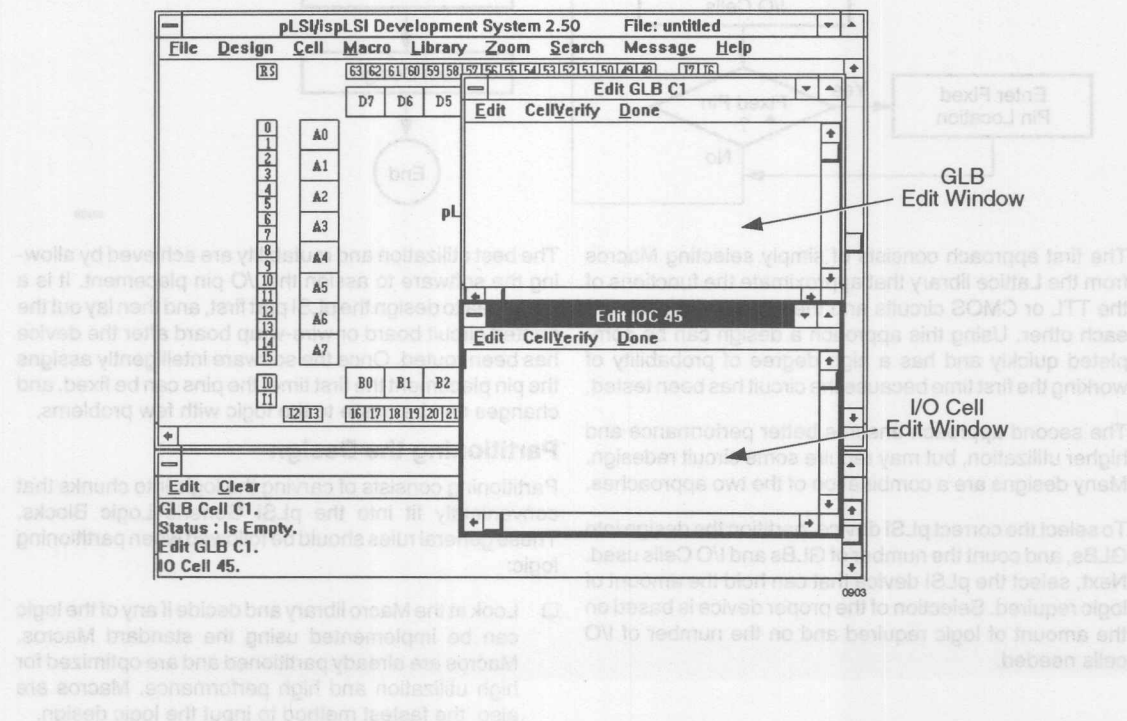
The method of entering the configuration data into a cell depends on what type of cell it is:

Configuration data for Generic Logic Blocks is entered using a combination of Boolean Equations or Macros from the Lattice Standard Library.

Configuration data for I/O Cells is entered using Macros only. There is a complete set of Macros which describes all possible combinations of input, output, and I/O cell configurations.

Configuration data for the Dedicated Input Pins and the Clock Input Pins is entered using a subset of the I/O Cell Macros. Because these pins are inputs only, and do not have input registers, many of the standard I/O Cell Macros cannot be used.

Figure 4. Open Edit Windows



The Design Flow

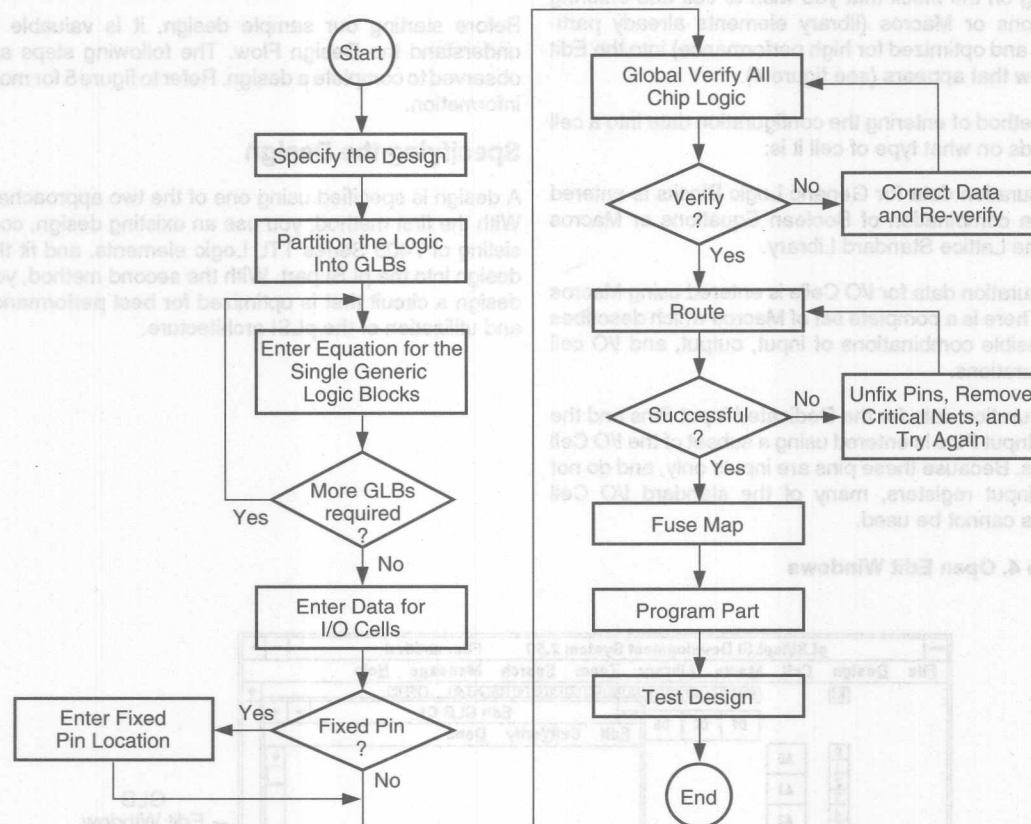
Before starting our sample design, it is valuable to understand the Design Flow. The following steps are observed to complete a design. Refer to figure 5 for more information.

Specifying the Design

A design is specified using one of the two approaches. With the first method, you use an existing design, consisting of 7400 Series TTL Logic elements, and fit the design into the pLSI part. With the second method, you design a circuit that is optimized for best performance and utilization of the pLSI architecture.

Beginner's Guide

Figure 5. Design Process Flow



The first approach consists of simply selecting Macros from the Lattice library that approximate the functions of the TTL or CMOS circuits and then connecting them to each other. Using this approach a design can be completed quickly and has a high degree of probability of working the first time because the circuit has been tested.

The second approach ensures better performance and higher utilization, but may require some circuit redesign. Many designs are a combination of the two approaches.

To select the correct pLSI device, partition the design into GLBs, and count the number of GLBs and I/O Cells used. Next, select the pLSI device that can hold the amount of logic required. Selection of the proper device is based on the amount of logic required and on the number of I/O cells needed.

The best utilization and routability are achieved by allowing the software to assign the I/O pin placement. It is a good idea to design the pLSI part first, and then lay out the printed circuit board or wire-wrap board after the device has been routed. Once the software intelligently assigns the pin placement the first time, the pins can be fixed, and changes can be made to the logic with few problems.

Partitioning the Design

Partitioning consists of carving the logic into chunks that conveniently fit into the pLSI Generic Logic Blocks. These general rules should be followed when partitioning logic:

- Look at the Macro library and decide if any of the logic can be implemented using the standard Macros. Macros are already partitioned and are optimized for high utilization and high performance. Macros are also the fastest method to input the logic design.

- ❑ Know the capabilities of the GLB. It has 18 Inputs and 4 Outputs. The GLB has 20 Product Terms (PTs) that are grouped together in groups of 4, 4, 5, and 7 PTs. The registers in the GLB share a common clock. The registers within the GLB also share a common Reset Product Term.
- ❑ When an output has been fixed to a specific I/O pin, the signal that is used to generate that output must be generated within the same Megablock.
- ❑ There is only one Output Enable signal per Megablock. Outputs which share a common Output Enable signal should be placed in the same Megablock (see figure 3).
- ❑ Signals that are related to each other, such as those used for counters, shift registers, etc., should be placed into the same Megablock. This is done to reduce routing congestion.

Compiling the Design

Compiling the design is done using the Lattice pDS Software and consists of four steps:

1. **Entering the design.** Boolean equations or Macros are entered into the various cells and blocks on the pLSI device using a built in text editor. After each cell has been entered, a *Local Verify* is done to check for syntactical or logical errors within that cell.
2. **Verifying the design.** This is done globally after all the design has been entered. This verification looks for such problems as inputs that are not connected to the GLBs or nets that have duplicate names. The design must completely pass a Global Verify before any of the following steps can happen.
3. **Routing the design.** This is the next step after a successful Verify. The Router interconnects the Generic Logic Block and I/O Cell inputs and outputs. The option of fixing certain input and output signals to specific device pins is available.
4. **Generating the Fusemap.** This takes the verified and routed design and creates the *JEDEC* (a standard binary fuse file) necessary to program the part. This is a modified format JEDEC file, and the file generated has a suffix of .JED.

Programming the Part

Once the design has been compiled, the next step is to program the part. This can either be done on the board if using in-system programming (ISP) or in a separate programmer. Using a separate programmer requires that the part be removed from the target system socket and inserted into the programmer to program the part.

Testing the Design

The last step in the process is testing the design. The design is tested by putting it on the board and seeing if it works correctly. If corrections need to be made, the appropriate GLBs or I/O Cells are reprogrammed, and the design is recompiled. Because the pLSI 1032 is an electrically erasable and reprogrammable part, the same part can be used again.

The Sample Design

The sample design is a simple one. We are going to design a 4-bit binary counter using Boolean equations and place it into a pLSI 1032 device. We will then take the design through the compilation process, generate a fuse file and program a part.

The counter has the following specifications:

- ❑ A 4-bit Synchronous Binary Counter.
- ❑ An Active High Cascade In (CI) and Cascade Out (CO) Pins.
- ❑ An Active High Count Enable (CE) Pin.
- ❑ A Synchronous Reset Pin.

Figure 6 shows the schematic diagram and Figure 7 shows the logic symbol for this counter. Because the counter has 5 outputs (Q0, Q1, Q2, Q3, and Cascade Out) it occupies two GLBs.

In this design example, the Clock and I/O pins are assigned to be compatible with the Lattice ispLSI 1032 Demonstration board. This allows the design to be tested easily.

The input signals Cascade In, Count Enable and Reset are connected to three bits of the 8-bit DIP switch, and the five outputs are connected to five of the discrete LED outputs.

Defining the Counter

In defining the counter, the first step is to write the equations. The equations for the 4-bit binary counter are expressed in Listing 1.

There are two inputs to the Exclusive-OR gate in front of the D input to the register. We shall call the one that receives its input from the feedback of the same register as the data input. It is to the left of the \oplus (XOR) symbol in the above equations. The other input is connected to the control terms Cascade In and Count Enable. These are called the control input. When the control input to the XOR is a zero the output of the XOR follows the data input (Hold.) When the control input is a one, the output of the XOR is inverted from the input (Increment.)

Beginner's Guide

Figure 6. Counter Schematic Diagram

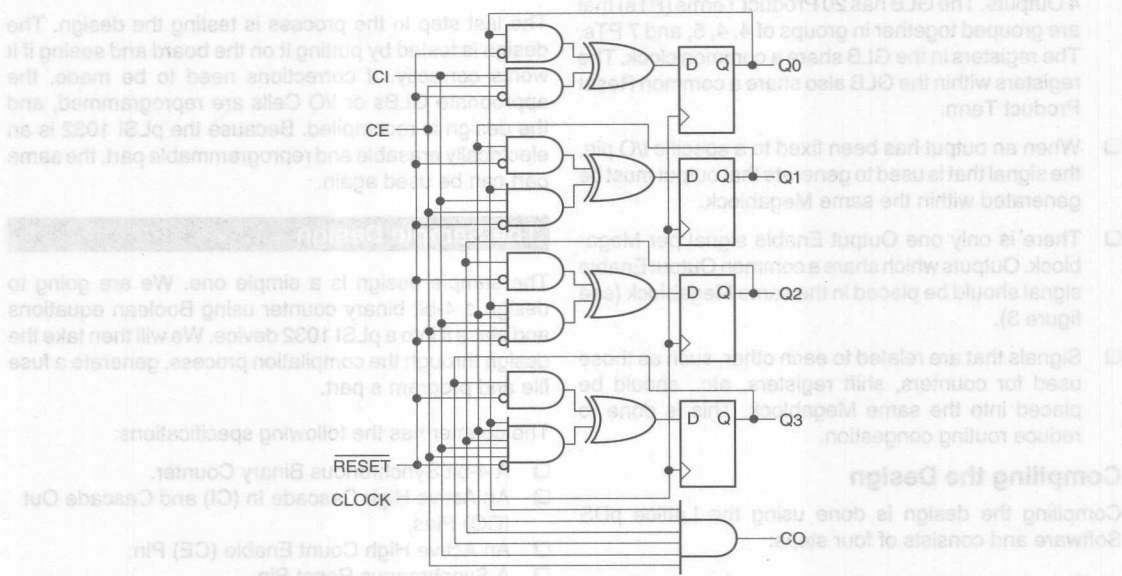
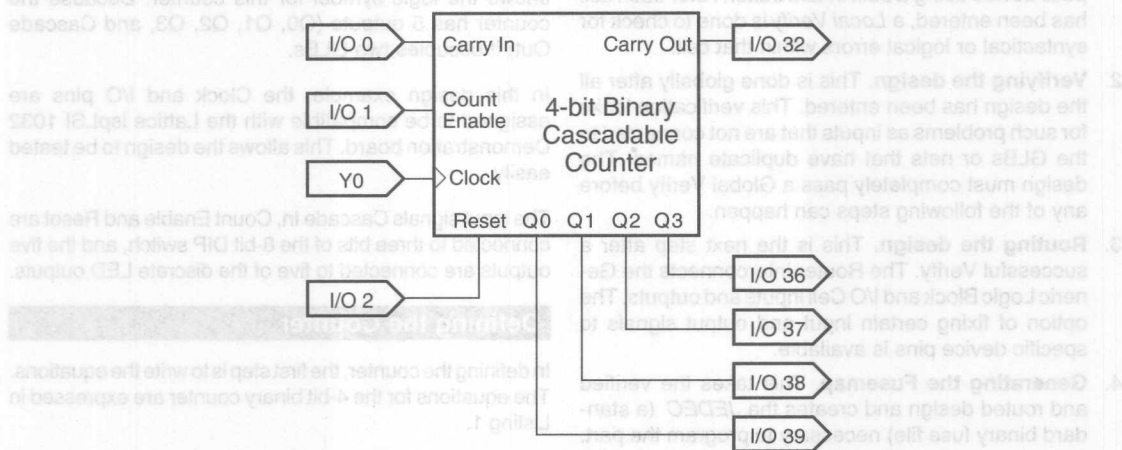


Figure 7. Sample Cascadable Counter Logic



When either Cascade In or Count Enable is low and RST is low, the Q0 output from the counter remains in its current state (Hold). When Cascade In and Count Enable are both high and RST is low, the Q0 output toggles on each successive clock (increment.) When RST goes High, the inputs to the Data side of the XOR gate and the Control side go low. This causes the output of the counter to go low on the next clock edge (Reset.)

Each successive stage operates similarly, except during transition, (Increment), when the outputs of all previous stages are at logic level one. The Carry Out signal is only generated when all the stages have reached a one and both Cascade In and Counter Enable are a one.

Once the equations have been defined, enter them into the GLBs. Follow these steps:

1. From within Windows, start the Lattice pDS Software by double clicking on the Lattice Icon.
2. When the Lattice software starts, it displays the block diagram of the pLSI 1032 part. Open GLB C1 for editing by double clicking on it. The edit window displays.
3. Enter the equations shown in Listing 2 into the edit window for GLB C1.
4. Verify the equations by clicking on the Cell Verify menu option. If errors appear in the Message window, find out what is wrong, and correct it. Things to look for are typing errors, missing semicolons, or incorrect symbols. Re-verify after making corrections.
5. Close the Edit window for GLB C1 by selecting the Done option from the Cell Edit Menu.
6. Open GLB C2 for editing by double clicking on it.
7. Enter the following equations into the edit window for GLB C2:

```
SIGTYPE CO OUT;
EQUATIONS
  CO = Q0 & Q1 & Q2 & Q3 & CI & CE;
END
```

Listing 1. Counter Equations

```
Q0 = (Q0 & !_RST) $$ (CI & CE & !_RST)
Q1 = (Q1 & !_RST) $$ (Q0 & CI & CE & !_RST)
Q2 = (Q2 & !_RST) $$ (Q0 & Q1 & CI & CE & !_RST)
Q3 = (Q3 & !_RST) $$ (Q0 & Q1 & Q2 & CI & CE & !_RST)
CO = Q0 & Q1 & Q2 & Q3 & CI & CE
```

Listing 2. GLB Equations

```
SIGTYPE Q0 REG OUT;
SIGTYPE Q1 REG OUT;
SIGTYPE Q2 REG OUT;
SIGTYPE Q3 REG OUT;
EQUATIONS
  Q0.CLK = _CLK;
  Q0 = (Q0 & !_RST) $$ (CI & CE & !_RST);
  Q1 = (Q1 & !_RST) $$ (Q0 & CI & CE & !_RST);
  Q2 = (Q2 & !_RST) $$ (Q0 & Q1 & CI & CE & !_RST);
  Q3 = (Q3 & !_RST) $$ (Q0 & Q1 & Q2 & CI & CE & !_RST);
END
```

8. Verify the equations by clicking on the Cell Verify menu option.

9. Close the Cell Edit window by clicking on the Done option in the menu bar. See figure 8.

At this point, the logic for the counter is completely specified, but we still must connect the Clock and the Inputs and Outputs.

10. Open Clock Input Y0 by double clicking on it. It may be necessary to Zoom in on the Clock area of the Logic Diagram to determine which pin is Y0.
11. Enter the following equations into the edit window for Clock Input Y0:


```
XPIN CLK X_CLK LOCK 20;
IB11 (_CLK, X_CLK );
```
12. Verify the equations by clicking on the Cell Verify menu option.
13. Once the cell verifies correctly, close the Cell Edit window by clicking on the Done option in the menu bar.

Beginner's Guide

14. Repeat Steps 10 through 13 for the Cascade In input pin located at I/O 0 using these equations:

```
XPIN IO XCI LOCK 26;
IB11 (CI, XCI);
```

15. Repeat Steps 10 through 13 for the Count Enable input pin located at I/O 1 using these equations:

```
XPIN IO XCE LOCK 27;
IB11 (CE, XCE);
```

16. Repeat Steps 10 through 13 for the Reset input pin located at I/O 2 using these equations:

```
XPIN IO X_RST LOCK 28;
IB11 (_RST, X_RST);
```

17. Repeat Steps 10 through 13 for the Q0 output pin located at I/O 39 using these equations:

```
XPIN IO XQ0 LOCK 75;
OB11 (XQ0, Q0);
```

18. Repeat Steps 10 through 13 for the Q1 output pin located at I/O 38 using these equations:

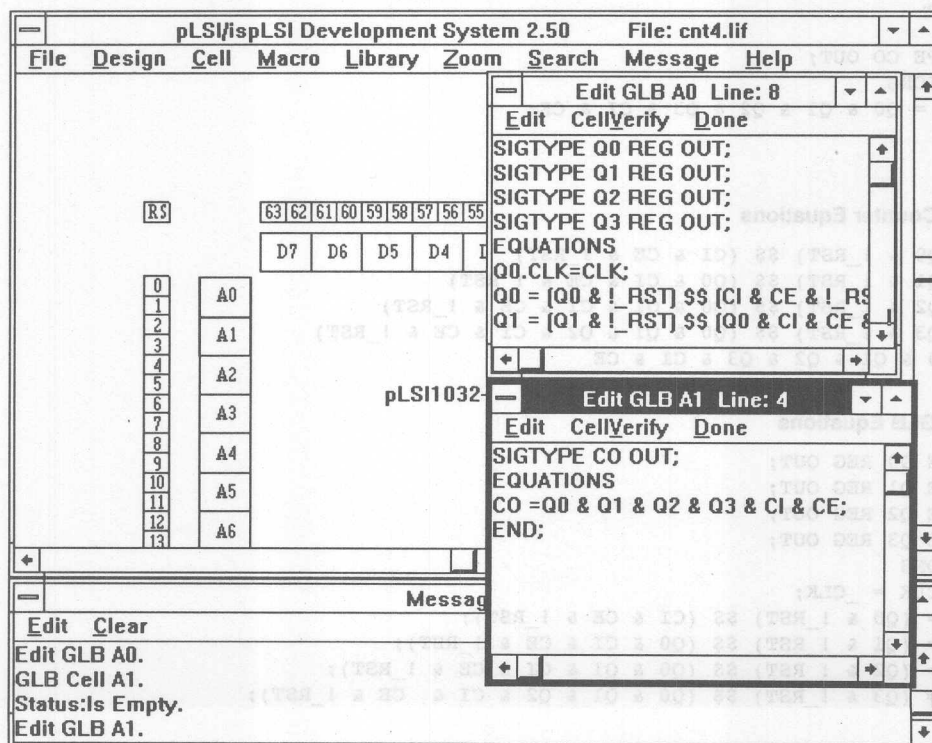
```
XPIN IO XQ1 LOCK 74;
OB11 (XQ1, Q1);
```

Note: With the Lattice pDS Software you can have two separate Edit Windows open at the same time. This means that you can Copy the equations from I/O Cell 39 and Paste them into I/O Cell 38. The data in both cells is similar, and you can use the Windows editing commands to make changes.

19. Repeat Steps 10 through 13 for the Q2 output pin located at I/O 37 using these equations:

```
XPIN IO XQ2 LOCK 73;
OB11 (XQ2, Q2);
```

Figure 8. Cell Entry Windows with Counter Equations.



20. Repeat Steps 10 through 13 for the Q3 output pin located at I/O 36 using these equations:

```
XPIN IO XQ3 LOCK 72;
OB11 (XQ3, Q3);
```

21. Repeat Steps 10 through 13 for the Carry Out output pin located at I/O 35 using these equations:

```
XPIN IO XCO LOCK 71;
OB11 (XCO, CO);
```

Now, the Inputs, Outputs and Clocks are connected, and the equations for the counter have been entered and verified. The design is complete and ready to be Globally Verified. Before proceeding, save the work.

22. From the Menu Bar, select the File Option, and choose Save As. The pDS Software prompts you for the name of the file that you are saving. Type in the name COUNTER. The suffix .LIF (Lattice Internal File) is automatically appended.

The next step in the development process is to Globally Verify the integrity of the design. Global Verify first performs a Cell Verify on GLBs or I/O Cells which have not already been verified, or which have changed since the last Cell Verification. Then it checks interconnections between the GLBs looking for problems such as outputs which are not used or inputs that are not connected.

23. From the Design Menu option in the Menu Bar, select Verify. This starts the Global verify process. If Verify finds any problems, it lists them in the Message window at the bottom of the screen. The verifier also creates a netlist file that the Router uses to route the design. Once the design passes verify, it is ready to be routed.

24. From the Design Menu option in the Menu Bar, select Route. This module places I/O pins that have not previously had their positions defined, and interconnects all the logic blocks and I/O cells on the device. When Route is invoked, a list of all the I/O pins displays. If you have not previously defined which signals are connected to which pins, this is the time to do it.

25. From within the Route Message Window, click on the Execute button. This starts the router. Routing is an entirely automatic process, and requires no intervention. As before, if any problems occur, they are listed in the message window.

26. The last step in the compilation process is to generate the Fusemap. This is accomplished by clicking on the Fusemap option in the design menu. Like Route, Fusemap is an entirely automatic process, and should require no intervention. The output from the Fusemap program is the .JED file and it is used to program the part.

The design is now complete. Because it was given a name previously (COUNTER.LIF) you can simply click on the Save command in the File menu to save the work. All that remains is to program the part and test the design.

Review of the Syntax

This is a brief review of the syntax used in the example design. For complete information see the Language Reference section of the Software Manual included with the Lattice pDS Software.

The operators that the Lattice pDS Software uses are similar to those used by the Data I/O ABEL program. The operators and an example of how they are used are shown in the table below. The Precedence of Evaluation is also shown where 1 is the highest precedence. See table showing Precedence of evaluation.

Table: Precedence of Evaluation

Operator	Precedence	Description	Example
!	1	NOT	!A
\$\$	2	XOR (XOR Gate in GLB)	A \$\$ B
&	3	AND	A & B
#	4	OR	A # B
\$	5	XOR (Soft)	A \$ B
!\$	5	XNOR (Soft)	A !\$ B

Beginner's Guide

Review of the Syntax (Continued)

In addition to the equations, there are several other lines that need to be included in the GLB or I/O Cell definition. They are:

SYM; The symbol line consists of five parts:

- The Keyword SYM that indicates what type of line this is to be.
- The Symbol Name. This is either GLB or IOC.
- The Cell location.
- The Symbol Level used by other software packages. For our purposes, always use a 1.
- The Symbol User Name. This is an assigned name that appears in the GLB or IOC in place of its location designation.

SIGTYPE; Used to define signal attributes within a GLB.

OUT defines a combinatorial output.

REG OUT defines a Registered Output.

EQUATIONS; Indicate the start of the Equation Section for a GLB or I/O Cell.

MACRO; Indicates the usage of a Macro Logic Element from the Macro Library.

END; Signifies the end of an Equation Section, a GLB or I/O Cell definition, a Declaration

Section, or a Macro Definition.

There can be more than one END statement in a GLB.

Comments are indicated by preceding the comment with two forward slashes:

// This is an example comment line.

Programming the Device

The Fusemap program generated a fuse (.JED) file which needs to be permanently programmed into an ispLSI or pLSI 1032 device. Programming the part is done using one of three methods:

- ☐ RS-232 Link Programmer for ispLSI or pLSI device
- ☐ In-system program for ispLSI device
- ☐ Motherboard Programmer for ispLSI device

For a programmer that is controlled by a serial RS-232 link, the Lattice pDS Software can call up the Windows Terminal Program. By using your PC to emulate a terminal, you can give the programmer the commands necessary to set it up to receive the .JED file. The Download command in the Windows Terminal program transfers the file to the programmer. Because the .JED file is an ASCII format, a text download is used.

An ispDOWNLOAD Cable is offered as an option with the pDS Software. The cable connects to the parallel port on a PC and controls the programming process. If the target system is designed to use in-system programming, the

part can be programmed right on the board or using the isp Engineering Kit.

For designers who need to integrate isp into their on board programming control using a microprocessor, Lattice provides ispCODE (C source code) to allow for customization of the isp user interface.

For programming the ispLSI 1032 part, follow these commands.

1. From the Design Menu select the Program Option. This invokes the In-system programming module.
2. The isp module prompts for the name of the JEDEC file. Click on COUNTER.JED in the file list and then click on OK. It may already have COUNTER.JED as the default file name. If this is so, then just click on OK.
3. Programming takes a few seconds. If any errors are encountered, they are listed in the message box.

When programming is complete, the part is reset and sent back into the operating mode. It can then be tested by applying the required inputs and looking at the outputs.

Advanced Design Concepts

Working with Macros

The Lattice pDS Software comes with a library of over 200 Macro logic elements. These logic blocks are similar to 7400 TTL logic. Some example Macros are listed in Table 1.

For complete information on the Macro Library refer to the Macro Reference Manual that comes with the Lattice pDS Software. In addition to using Macros from the Lattice library, you can either create custom Macros from scratch, or modify Macros from the Lattice library to satisfy design requirements.

We are going to take a Macro from the library that is identical to the counter just created, and cascade it with the counter. The Macro element that is used to do this is named CBU24. The schematic diagram is shown in figure 9.

1. Read in the previous design using the File Open command. The name of the file is COUNTER.LIF.
2. Choose the Library menu option and highlight **Select** to invoke the library window. Click on the System Lib button and then click on OK.
3. Invoke the Macro window by clicking on **MACRO** in the Menu Bar.
4. Select the Macro *CBU24* from the list of Macros.
5. Click on GLB C3 to Select it.
6. Click on the *PLACE* command in the Macro Menu. This places the first half of the 4-bit counter Macro in GLB C3. The signal names that were placed in the GLB are the default signal names, and need to be changed to correspond to the signal names that used so that the router is able to connect them.
7. The original text in the cell was:
`CBU24_2(CAO,[Q0..Q3],CAI,EN);`
Change that to read:
`CBU24_2(CAO,[Q4..Q7],CO,CE);`
The default signal names are changed to match those already used in as shown in Table 2.
8. Perform a Cell Verify to ensure that no errors were introduced.
9. Click on *DONE* to close that GLB.
10. The Macro occupies two GLBs, so the second half of the Macro now needs to be placed. Click on *GLB C4* to place the second half.

Table 1. Macro Logic Element Examples

Macro Name	7400 Part Equivalent	Description	Number of GLBs Used
AND2	7408	2 Input AND Gate	1/4
XOR2	7486	2 Input Exclusive OR Gate	1/4
FJK21	74112	J-K Flip-Flop with Asynchronous Clear	1/4
CBU34	74161	4-Bit Preloadable Binary Counter with Reset	1 1/4
BIN27	74247	BCD to 7 Segment Decoder	2
SRR38	74166	8-Bit Parallel In-Serial Out Shift Register	2
ADDF4	74283	4-Bit Full Adder with Look Ahead Carry	4 3/4

Table 8- 0002

Beginner's Guide

Figure 9. Custom Binary Counter Cascaded with a Standard Macro Counter

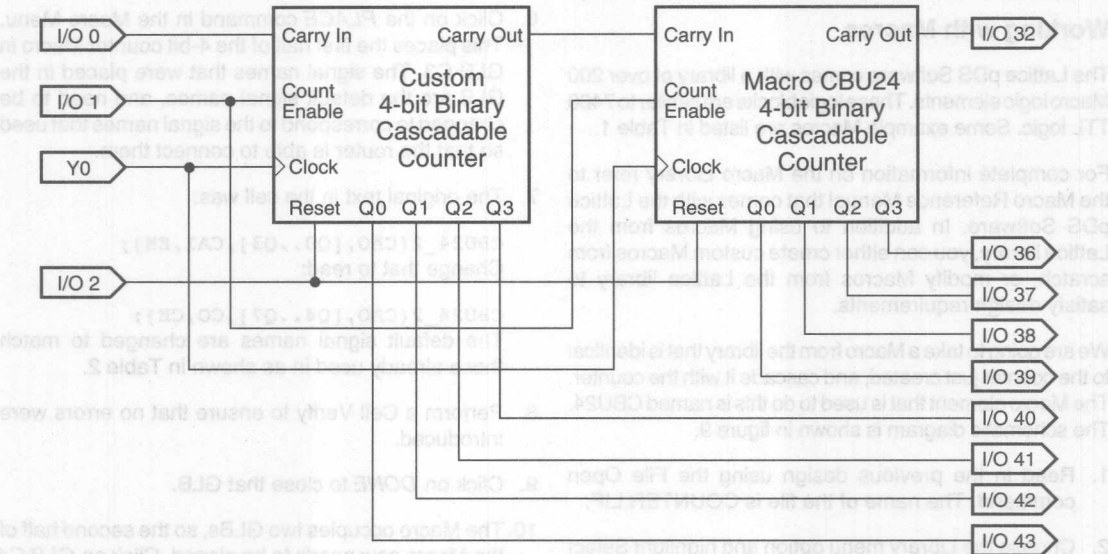


Table 2. Default Signal Names

Default	Signal	Is	Notes
CAO	Cascade Out	CAO	This is a new signal. We can use the default name.
Q0..Q3	Counter Outputs	Q4..Q7	We used Q0 through Q3 in the first counter. We need to assign new names so the router will not get confused.
CAI	Cascade In	CO	CO is the name that we assigned to the Cascade Out pin on the counter that we designed.
EN	Enable	CE	We called our Enable pin CE (Count Enable). This comes from a pin external to the device.

11. As before, the signal names that were placed in the GLB were the default names. They also need to be edited. The Lattice software placed the following code into the cell:

```
CBU24_1([Q0..Q3],CAI,CLK,EN,CD);  
Change it to read:
```

```
CBU24_1([Q4..Q7],CO,_CLK,CE,RST);  
As before, we have changed the default signal names to match those that we are already using. See Table 8-3.
```

12. As before, perform a *CELL VERIFY*, and click on *DONE* when through.

The counter has now been placed, and the inputs connected, but the outputs are still floating. Connect them to the I/O Cells as you did with the previous counter.

13. Select the Macro called *OB11* from the Macro list.

14. Click on IO Cell #40 to select it.

15. Click on *PLACE* in the Macro window. This configures I/O Cell #40 as an output buffer, but it used the default signal names. The text that was placed in the cell was:

```
OB11 (X00,A0);
```


You should change it to read:

```
OB11 (X04,Q4);
```

Q4 is the name of the first output of the counter. XO0 was changed to XO4 so that there would not be duplicate I/O cell names when we place the next cells.

15. Click on IO Cell #41 to select it.

16. Click on *PLACE* in the Macro window. Change the default signal names to match those used in your design:

```
OB11 (X00,A0);
```

Becomes:

```
OB11 (X05,Q5);
```

17. Use the same technique to connect I/O cell #42 to counter output Q6.

18. Use the same technique to connect I/O cell #43 to counter output Q7.

All the outputs are now connected, and the design is complete. As in the first design, you now need to do a *Global Verify* on the design, *Route* the nets and generate the *Fusemap*. You can see from this exercise how much simpler it is to complete a design when using Macros.

The use of Macros is not limited to those in the Lattice Macro library. Sometimes the standard Macro is close to, but not exactly what you need. You can copy any of the standard Lattice soft Macros into a personal library, and modify them to meet specific needs. You can also create Macros using Boolean equations and save them in your personal library for future use.

Conclusion

We have tried to give a feeling of how to design using pDS Software from definition to completion. In this Beginner's Guide, we:

- ☐ Looked at the Lattice pDS Software and its various elements.
- ☐ Explained the design flow from beginning to end.
- ☐ Looked at the syntax needed for entering a design.
- ☐ Defined a small counter and partitioned it into GLBs.
- ☐ Entered the design for that counter into the development system.
- ☐ Took that design through the compilation process. (Verify, Route, and Fusemap).
- ☐ Programmed a part.
- ☐ Tested the design.
- ☐ Changed the design, and introduced the use of Macros.
- ☐ Recompiled that design and tested it.

From this you can see how simple it is to design using the Lattice ispLSI or pLSI families. If you have followed all of these steps, then you are ready to complete a design of your own.

Table 3. Renaming Default Signal Names

Default	Signal	Is	Notes
Q0..Q3	Counter Outputs	Q4..Q7	We used Q0 through Q3 in the first counter. We need to assign new names so the router will not get confused.
CAI	Cascade In	CO	CO is the name that we assigned to the Cascade Out pin on the counter that we designed.
CLK	Clock	_CLK	We named the signal that we brought in on pin Y0 _CLK
EN	Enable	CE	We called our Enable pin CE (Count Enable). This comes in from pin 27.
CD	Clear Direct	RST	Our reset signal was brought in on pin 28 and called RST.

Conclusion

We have tried to give a feeling of how to design using QDS Software from definition to completion. In this Beginner's Guide, we:

- Looked at the Lattice QDS Software and its various elements.
- Explained the design flow from beginning to end.
- Looked at the syntax needed for entering a design.
- Defined a small counter and partitioned it into GLEs.
- Entered the design for that counter into the development system.
- Took that design through the compilation process (Verity, Route, and Fusemap).
- Programmed a part.
- Tested the design.
- Changed the design, and introduced the use of Macros.
- Recompiled that design and tested it.

From this you can see how simple it is to design using the Lattice JapLSI or JLSI families. If you have followed all of these steps, then you are ready to complete a design of your own.

- You should change it to read:
- 0B11 (X04,04):
04 is the name of the first output of the counter. X04 was changed to X04 so that there would not be duplicate I/O cell names when we place the next cells.
15. Click on IO Cell #41 to select it.
16. Click on PLACE in the Macro window. Change the default signal names to match those used in your design.
- 0B11 (X04,04):
Becomes:
0B11 (X02,05):
17. Use the same technique to connect I/O cell #42 to counter output 08.
18. Use the same technique to connect I/O cell #43 to counter output 07.
- All the outputs are now connected, and the design is complete. As in the first design, you now need to do a Global Verity on the design, Route the nets and generate the Fusemap. You can see from this exercise how much simpler it is to complete a design when using Macros.
- The use of Macros is not limited to those in the Lattice Macro library. Sometimes the standard Macro is close to, but not exactly what you need. You can copy any of the standard Lattice soft Macros into a personal library, and modify them to meet specific needs. You can also create Macros using Boolean equations and save them in your personal library for future use.

Table 3. Renaming Default Signal Names

Default	Signal	is	Notes
00_02	Counter Outputs	04_07	We used 00 through 03 in the first counter. We need to assign new names so the counter will not get confused.
0A1	Cascade In	00	00 is the name that we assigned to the Cascade Out pin on the counter that we designed.
CLK	Clock	_CLK	We named the signal that we brought in on pin Y0 _CLK.
EN	Enable	CE	We called our Enable pin CE (Count Enable). This comes in from pin X7.
00	Clear Direct	RST	Our reset signal was brought in on pin Z8 and called RST.

Introduction

As high density programmable logic becomes more common place, determining exactly which functions to integrate and how to integrate these functions becomes more challenging. Some of the obvious considerations when integrating a design include speed and density. Beyond these concerns several other design and system details must be evaluated. In the following example, these design details will be examined and fully addressed. Design considerations can be broken into the following hierarchy: 1) System considerations including technology, reliability, and testability. 2) Design considerations which include partitioning a design for a specific architecture, determining I/O, and speed concerns. 3) Integration of the design into an ispLSI device. This includes utilizing the ispLSI and pLSI architecture for the best speed and efficient random logic consolidation.

A Dual Processor Controller

The design shown in figure 1 is a dual processor controller which sits on a backplane bus to which other CPUs have access. All of the CPUs communicate via the backplane bus by sending interrupts back and forth. This design also contains an independent 32-bit general purpose counter along with CPU control logic for memory and I/O.

Before partitioning the design, one must consider board space and reliability. For example, in some systems where board space and reliability are at a premium, it

may be desirable to surface mount all components. In these cases, using sockets may be necessary to minimize manufacturing problems for the programmable devices. Of course, all Lattice ispLSI devices are in-system programmable, so removing devices from the board is not necessary if reprogramming is required. Another benefit to directly soldering components on the board is that less board space is needed and less capacitance will load the outputs. Therefore, soldering devices directly on the board will not increase propagation delay. To reprogram the ispLSI device, 5 volts and a five wire interface are all that is needed. In addition, choosing an instantly reprogrammable technology allows complete testability. Lattice tests for and guarantees 100% AC, DC, functional and programming yields.

Having considered these overall issues, we can now look at partitioning the design. Many designers partition by using GAL devices or other PLDs for speed and fast state machine control, and FPGAs for interface and random logic. The Lattice ispLSI family rewrites these basic design rules. With the Lattice ispLSI and pLSI families of high density programmable logic devices, the designer acquires speed and density in one device! The design must still be partitioned, but within the Generic Logic Blocks of the Lattice ispLSI device instead of between several discrete PLDs and/or FPGAs.

This design can be broken up into three major blocks: the two interrupt and bus random logic blocks, the data block consisting of a 32-bit counter with a 32-to-16 multiplexer, and the memory and I/O control logic state machine.

Figure 1. Dual Process Controller Block Diagram

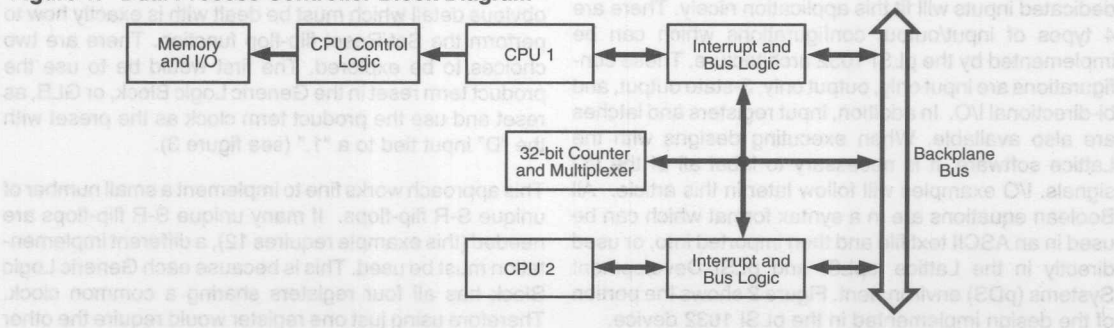
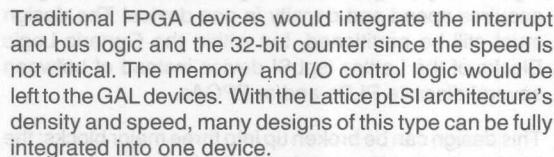


Figure 2. The Partitioned Design



Interrupt and Bus Random Logic

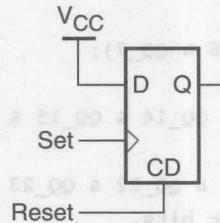
Let us examine the details of each of the three sections to see how they would be implemented into the pLSI architecture. First, how to implement the decoding and latching of the interrupts. For this design, integration of the decoding logic for the set and reset terms and set/reset flip-flops is necessary.

The decoding logic is easily integrated, because the architecture has the familiar AND-OR structure. The less obvious detail which must be dealt with is exactly how to perform the Set/Reset flip-flop function. There are two choices to be explored. The first would be to use the product term reset in the Generic Logic Block, or GLB, as reset and use the product term clock as the preset with the "D" input tied to a "1." (see figure 3).

This approach works fine to implement a small number of unique S-R flip-flops. If many unique S-R flip-flops are needed (this example requires 12), a different implementation must be used. This is because each Generic Logic Block has all four registers sharing a common clock. Therefore using just one register would require the other

ispLSI and pLSI: A Multiple Function Solution

Figure 3. D-Type flip-flop Configured as an S-R flip-flop



three outputs to be combinatorial, or registers with the same clock and reset. If there are several unique S-R flip-flops, each would have to exist in separate GLBs. This is not an efficient use of the architecture, unless the other outputs can be used as combinatorial logic. For this example, a more effective use of the GLBs can be achieved by making an S-R flip-flop from gates. The logic equations necessary are shown in Listing 1.

With this implementation, two S-R registers can fit into one GLB. The limiting factor in deciding whether two registers will fit, is the number of inputs necessary to perform the S-R function. Each GLB has a maximum of 18 inputs. If the number of inputs (including fast feedbacks), for the two registers is 18 or less, then both equations can be used in one GLB. In this design we have a total of 12 S-R registers. Listing 1 shows the equations for two S-R registers from the design, followed by the same equations reconfigured using the gate S-R implementation in Listing 2.

The number of unique input and feedback signals in the 4 equations above, is 14. Since this is less than 18, the

equations will fit in one GLB. To implement the other 10 S-R registers, simply use the same strategy and partition the logic into five other GLBs.

Data Path: 32-bit Counter and 32-to-16 Multiplexer

The next task is deciding how to build the 32-bit counter and the 32-to-16 multiplexer data latch. Using the ispLSI architecture, counter implementations up to 16-bits are straightforward. Up to 16-bits, the counter can run at the full speed of the device. Two reasons the counter is able to execute at full speed are: 1) the wide input GLBs, and 2) T-type flip-flops configurable in the architecture. The T-type flip-flop is created by inserting an XOR gate before a D-type flip-flop and feeding back the D output into one of the two inputs to the XOR gate. The other input to the XOR gate becomes the T-type flip-flop input. Beyond 16-bits, a counter must be cascaded into another level of logic because the total number of inputs needed exceeds the maximum allowed by the GLB architecture. Recall that each GLB has an 18 input limit. Two of the inputs are dedicated input pins and the other 16 are I/O pins or fast feedbacks. Therefore, to implement a 32-bit counter, we must use two more GLBs to decode the point at which the counter has reached the full 16-bit mark. This is accomplished by setting an output true when all bits (0 - 15) are a "1." Also, it is necessary to decode the point at which the counter has reached the full 24-bit mark. This is done by setting an output true when all bits (0-23) are a "1." Using these intermediate terminal count outputs, a 32-bit counter can be implemented in 10 GLBs. This 32-bit counter can run at 40 MHz as implemented here, or up to 80 MHz if the carry out is pipelined. The equations for this counter are shown in Listing 3.

Listing 1.

```
Q = !Set # !Qbar;      // Q is the output of the S-R flip-flop
Qbar = !Reset # !Q;    // Qbar is the inversion of Q
```

Listing 2.

```
reset1 = bp_int_clr & bp_data12 # bp_reset;
reset2 = bp_int_clr & bp_data11 # bp_reset;
set1 = !m_as & !ipc_int & mdata8 & !mdata10 & !mdata11 & !mdata12;
set2 = !m_as & !ipc_int & mdata8 & mdata10 & !mdata11 & !mdata12;
These equations are now optimized to combine the logic in one GLB:
Q1 = !(!m_as & !ipc_int & mdata8 & !mdata10 & !mdata11 & !mdata12) # !Q1bar; //
!set1
Q1bar = !(bp_int_clr & bp_data12 # bp_reset) # !Q1; //!reset1
Q2 = !(!m_as & !ipc_int & mdata8 & mdata10 & !mdata11 & !mdata12) # !Q2bar; //
!set2
Q2bar = !(bp_int_clr & bp_data11 # bp_reset) # !Q2; // !reset2
```


ispLSI and pLSI: A Multiple Function Solution

Listing 3.

```
// 0-7 decode
TC_1 = (QQ_0 & QQ_1 & QQ_2 & QQ_3 & QQ_4 & QQ_5 & QQ_6 & QQ_7);
// 0-15 decode
TC_2 = (QQ_8 & QQ_9 & QQ_10 & QQ_11 & QQ_12 & QQ_13 & QQ_14 & QQ_15 & TC_1);
// 0-23 decode
TC_3 = (QQ_16 & QQ_17 & QQ_18 & QQ_19 & QQ_20 & QQ_21 & QQ_22 & QQ_23 & TC_2);
// The QQ_0 to QQ_31 signals are the 32 counter output bits.
QQ_0 = QQ_0 $$ VCC ;
QQ_1 = QQ_1 $$ QQ_0;
QQ_2 = QQ_2 $$ QQ_1 & QQ_0 ;
QQ_3 = QQ_3 $$ QQ_2 & QQ_1 & QQ_0 ;
QQ_4 = QQ_4 $$ QQ_3 & QQ_2 & QQ_1 & QQ_0 ;
QQ_5 = QQ_5 $$ QQ_4 & QQ_3 & QQ_2 & QQ_1 & QQ_0 ;
QQ_6 = QQ_6 $$ QQ_5 & QQ_4 & QQ_3 & QQ_2 & QQ_1 & QQ_0 ;
QQ_7 = QQ_7 $$ QQ_6 & QQ_5 & QQ_4 & QQ_3 & QQ_2 & QQ_1 & QQ_0 ;
QQ_8 = QQ_8 $$ TC_1 ;
QQ_9 = QQ_9 $$ QQ_8 & TC_1 ;
QQ_10 = QQ_10 $$ QQ_9 & QQ_8 & TC_1 ;
QQ_11 = QQ_11 $$ QQ_10 & QQ_9 & QQ_8 & TC_1 ;
QQ_12 = QQ_12 $$ QQ_11 & QQ_10 & QQ_9 & QQ_8 & TC_1 ;
QQ_13 = QQ_13 $$ QQ_12 & QQ_11 & QQ_10 & QQ_9 & QQ_8 & TC_1 ;
QQ_14 = QQ_14 $$ QQ_13 & QQ_12 & QQ_11 & QQ_10 & QQ_9 & QQ_8 & TC_1 ;
QQ_15 = QQ_15 $$ QQ_14 & QQ_13 & QQ_12 & QQ_11 & QQ_10 & QQ_9 & QQ_8 & TC_1 ;
QQ_16 = QQ_16 $$ TC_2 ;
QQ_17 = QQ_17 $$ QQ_16 & TC_2 ;
QQ_18 = QQ_18 $$ QQ_17 & QQ_16 & TC_2 ;
QQ_19 = QQ_19 $$ QQ_18 & QQ_17 & QQ_16 & TC_2 ;
QQ_20 = QQ_20 $$ QQ_19 & QQ_18 & QQ_17 & QQ_16 & TC_2 ;
QQ_21 = QQ_21 $$ QQ_20 & QQ_19 & QQ_18 & QQ_17 & QQ_16 & TC_2 ;
QQ_22 = QQ_22 $$ QQ_21 & QQ_20 & QQ_19 & QQ_18 & QQ_17 & QQ_16 & TC_2 ;
QQ_23 = QQ_23 $$ QQ_22 & QQ_21 & QQ_20 & QQ_19 & QQ_18 & QQ_17 & QQ_16 & TC_2 ;
QQ_24 = QQ_24 $$ TC_3 ;
QQ_25 = QQ_25 $$ QQ_24 & TC_3 ;
QQ_26 = QQ_26 $$ QQ_25 & QQ_24 & TC_3 ;
QQ_27 = QQ_27 $$ QQ_26 & QQ_25 & QQ_24 & TC_3 ;
QQ_28 = QQ_28 $$ QQ_27 & QQ_26 & QQ_25 & QQ_24 & TC_3 ;
QQ_29 = QQ_29 $$ QQ_28 & QQ_27 & QQ_26 & QQ_25 & QQ_24 & TC_3 ;
QQ_30 = QQ_30 $$ QQ_29 & QQ_28 & QQ_27 & QQ_26 & QQ_25 & QQ_24 & TC_3 ;
QQ_31 = QQ_31 $$ QQ_30 & QQ_29 & QQ_28 & QQ_27 & QQ_26 & QQ_25 & QQ_24 & TC_3 ;
```


ispLSI and pLSI: A Multiple Function Solution

The 32-to-16 multiplexer latch is the next logic block to be constructed. In this design, the multiplexer allows the system bus access to 16-bits of the counter at a time. Either the high 16-bits (16-31) or the low 16-bits (0-15) are enabled to the bus. Since this multiplexer latch is a simple OR gate control function into a register, these 16-bits can be placed into 4 GLBs. Recall that each GLB has a maximum of four outputs. The equations for one GLB are shown in listing 4.

These 16-bits are also 3-stated by a control pin. In the ispLSI 1032 architecture, 4 unique output enable terms are allowed. Each output enable can control up to 16 outputs or bi-directional pins. For example, a design could have 64 3-state outputs, but 4 output enable control signals would be used to control 16 outputs each. It is important to note that if an output enable signal is to control more than 16 outputs, the output enable signal will need to be defined more than once. In this design

only 16 outputs are controlled by one output enable signal, therefore only one output enable is used. This signal is provided by defining the output enable in a GLB as shown in listing 5.

Memory and I/O State Machine

Considering the memory and I/O state machine and decoding logic, the ispLSI GLB architecture has a path which is optimal for decoding logic. This path is utilized by choosing the 4 product term bypass mode. This mode allows an output with 4 product terms or less to exhibit input pin to output pin propagation delays of no more than 15 ns! Since decoding logic typically uses 4 product terms or less, this mode can be used for the critical propagation delay paths. The designer is cautioned to use the 4 Product Term Bypass Mode sparingly, because too many paths designated as critical in any one design may result in a failure of the Place and Route algorithm. The syntax necessary to invoke the 4 product term bypass mode is shown in listing 6.

Listing 4.

```
OMDATA0I.CLK = CLK;
OMDATA0I = (!CNTELO & QQ_16 ) # (CNTELO & QQ_0 ); //select high or low word
OMDATA1I = (!CNTELO & QQ_17 ) # (CNTELO & QQ_1 );
OMDATA2I = (!CNTELO & QQ_18 ) # (CNTELO & QQ_2 );
OMDATA3I = (!CNTELO & QQ_19 ) # (CNTELO & QQ_3 );
*
*
*
OMDATA31I = (!CNTELO & QQ_31 ) # (CNTELO & QQ_15);
```

Listing 5.

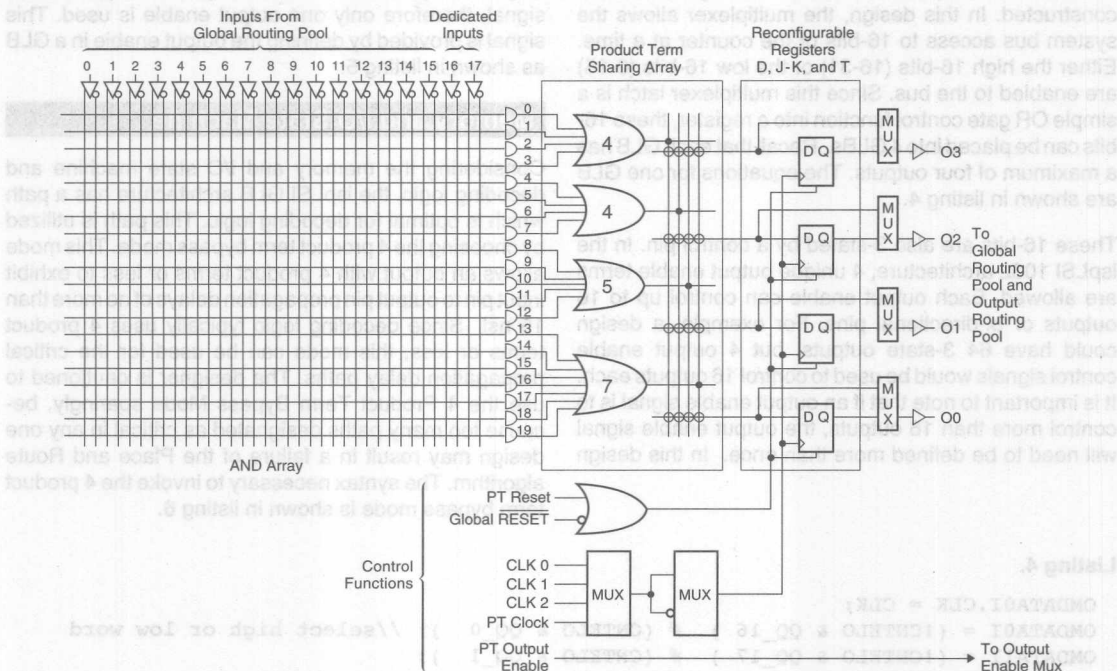
```
BP_INT_RDI.OE = BP_INT_RD0; //PROGRAMMABLE OUTPUT ENABLE SIGNAL
```

Listing 6.

```
SIGTYPE IO_SELECT 1 CRIT; // CRIT - TELLS THE SOFTWARE TO USE THE 4 PRODUCT
TERM // BYPASS MODE FOR THIS COMBINATORIAL OUTPUT
EQUATIONS
IO_SELECT = MA23 & MPA22 & !MPA21 & MPIO_MEM & !MPWR_RD #
MPA23 & !MPA22 & MPA21 & MPIO_MEM & MPWR_RD;
END;
```

ispLSI and pLSI: A Multiple Function Solution

Figure 4. GLB Product Term Sharing Array



The ispLSI 1032 is ideal for state machine applications because of two specific features. First, the I/O cell can be used to register or latch input signals. This attribute gives designers assurance that setup times to GLB registers will not be violated and metastability concerns are greatly diminished.

The second feature which configures efficient state machines is the standard GLB configuration with 4, 4, 5 and 7 product terms (see figure 4). The product terms can be tied together to perform wider product term functions which are always needed for complex state machines. For example, in a state machine which has an output consisting of 9 product terms, the architecture will allow 4 of the product terms to be tied to 5 additional product terms, to add up to the total of 9, which is required by the state machine output. Any configuration of product term grouping is possible, including all twenty! That's right, if the design needs twenty product terms for one output, this is handled in one pass through just one GLB.

The key to successfully implementing state machines into the ispLSI 1032 is to utilize the 18 maximum inputs with up to 4 outputs, and the ability to tie the 20 product terms together. Intelligent use of these features permit the designer to streamline state machine design.

Conclusion

As can be illustrated from the above discussion, the ispLSI architecture provides designers with unparalleled flexibility, density and speed. ispLSI devices are dense and flexible enough to incorporate random logic. The architecture also contains 18-wide inputs and XOR capability in each GLB which enable counters to be effortlessly implemented. The 4 product term bypass mode allows designers to successfully realize high speed applications. Finally, the ability to tie product terms together along with the input registers available at each I/O pin make this device ideal for state machine designs.

The complete Lattice Design File containing the Boolean equations for this design appears on the following pages.

ispLSI and pLSI: A Multiple Function Solution

Design LDF Listing

```
// tedfulla.ldf generated using Lattice pDS Version 2.50

LDF 1.00.00 DESIGNLDF;
DESIGN cdx_design 1.00;
PART pLSI 1032-90LJ;
DECLARE
END; //DECLARE

SYM GLB A4 1 INTA52; // Here are 2 S-R flip-flops
                        // OUT signifies a combinatorial output

SIGTYPE INTA2I OUT;
SIGTYPE INTA3I OUT;
SIGTYPE INTA2IBAR OUT;
SIGTYPE INTA3IBAR OUT;
SIGTYPE BP_INT_RDI OE; // OE signifies Output Enable
EQUATIONS
BP_INT_RDI = BP_INT_RDO;
INTA2I      = !(MAS & !IPC_INTI & !MDATA12I & !MDATA11I & !MDATA10I & MDATA8I)
            # !INTA2IBAR.PIN;
INTA2IBAR   = !INTA2I.PIN
            # !(BP_INT_CLRI & BP_DATA10I RSETI);
INTA3I      = !(MAS & !IPC_INTI &
            # !MDATA12I & !MDATA11I & MDATA10I & MDATA8I)
            # !INTA3IBAR.PIN;
INTA3IBAR   = !INTA3I.PIN
            # !(BP_INT_CLRI & BP_DATA15I RSETI);
END;
END;

SYM GLB A5 1 INTA52;
SIGTYPE INTA4I OUT;
SIGTYPE INTA5I OUT;
SIGTYPE INTA4IBAR OUT;
SIGTYPE INTA5IBAR OUT;
EQUATIONS
INTA4I      = !(MAS & !IPC_INTI & !MDATA12I & !MDATA11I & !MDATA10I & MDATA8I)
            # !INTA4IBAR.PIN;
INTA4IBAR   = !INTA4I.PIN
            # !(BP_INT_CLRI & BP_DATA14I RSETI);
INTA5I      = !(MAS & !IPC_INTI & !MDATA12I & !MDATA11I & MDATA10I & MDATA8I)
            # !INTA5IBAR.PIN;
INTA5IBAR   = !INTA5I.PIN
            # !(BP_INT_CLRI & BP_DATA13I RSETI);
END;
END;
```

ispLSI and pLSI: A Multiple Function Solution

```
SYM GLB B3 1 INTAMP45;
SIGTYPE INTAMP4I OUT;
SIGTYPE INTAMP5I OUT;
SIGTYPE INTAMP4IBAR OUT;
SIGTYPE INTAMP5IBAR OUT;
EQUATIONS
INTAMP4I = !(MAS & IPC_INTI & MDATA12I & MDATA11I & MDATA10I & MDATA8I)
# INTAMP4IBAR.PIN;
INTAMP4IBAR = !INTAMP4I.PIN
# !(MP_INT_CLRI & MP_DATA14I RSETI);
INTAMP5I = !(MAS & IPC_INTI & MDATA12I & MDATA11I & MDATA10I & MDATA8I)
# INTAMP5IBAR.PIN;
INTAMP5IBAR = !INTAMP5I.PIN
# !(MP_INT_CLRI & MP_DATA13I RSETI);
END;
END;

SYM GLB B4 1 INTAMP23;
SIGTYPE INTAMP2I OUT;
SIGTYPE INTAMP3I OUT;
SIGTYPE INTAMP2IBAR OUT;
SIGTYPE INTAMP3IBAR OUT;
SIGTYPE MP_INT_RDI OE;
EQUATIONS
MP_INT_RDI = MP_INT_RDO;
INTAMP2I = !(MAS & IPC_INTI & MDATA12I & MDATA11I & MDATA10I & MDATA8I)
# INTAMP2IBAR.PIN;
INTAMP2IBAR = !INTAMP2I.PIN
# !(MP_INT_CLRI & MP_DATA10I RSETI);
INTAMP3I = !(MAS & IPC_INTI & MDATA12I & MDATA11I & MDATA10I & MDATA8I)
# INTAMP3IBAR.PIN;
INTAMP3IBAR = !INTAMP3I.PIN
# !(MP_INT_CLRI & MP_DATA15I RSETI);
END;
END;

SYM GLB B5 1 INTAMP01;
SIGTYPE INTAMP0I OUT;
SIGTYPE INTAMP1I OUT;
SIGTYPE INTAMP0IBAR OUT;
SIGTYPE INTAMP1IBAR OUT;
EQUATIONS
INTAMP0I = !(MAS & IPC_INTI & MDATA12I & MDATA11I & MDATA10I & MDATA8I)
# INTAMP0IBAR.PIN;
INTAMP0IBAR = !INTAMP0I.PIN
# !(MP_INT_CLRI & MP_DATA12I RSETI);
INTAMP1I = !(MAS & IPC_INTI & MDATA12I & MDATA11I & MDATA10I & MDATA8I)
# INTAMP1IBAR.PIN;
INTAMP1IBAR = !INTAMP1I.PIN
# !(MP_INT_CLRI & MP_DATA11I RSETI);
END;
END;
```

ispLSI and pLSI: A Multiple Function Solution

```

SYM GLB B6 1 INTA01;
SIGTYPE INTA0I OUT;
SIGTYPE INTA1I OUT;
SIGTYPE INTA0IBAR OUT;
SIGTYPE INTA1IBAR OUT;
EQUATIONS
INTA0I      = !(MAS & !IPC_INTI & !MDATA12I & !MDATA11I & !MDATA10I & MDATA8I)
              # !INTA0IBAR.PIN;
INTA0IBAR   = !INTA0I.PIN
              # !(BP_INT_CLRI & BP_DATA12I RSETI);
INTA1I      = !(MAS & !IPC_INTI & !MDATA12I & !MDATA11I & MDATA10I & MDATA8I)
              # !INTA1IBAR.PIN;
INTA1IBAR   = !INTA1I.PIN
              # !(BP_INT_CLRI & BP_DATA11I RSETI);
END;
END;

SYM GLB A7 1 BPIPLS;
SIGTYPE BP_IPL0I OUT;
SIGTYPE BP_IPL1I OUT;
SIGTYPE BP_IPL2I OUT;
EQUATIONS
BP_IPL0I    = !INTA0I & !INTA2I & !INTA4I & BP_NMII & !DSP_INTI
              # !INTA0I & !INTA2I & !INTA4I & OS_TICKI & BP_NMII
              # BP_NMII & !TMS_INTI;
BP_IPL1I    = !INTA1I & !INTA3I & !INTA5I & OS_TICKI & BP_NMII & TMS_INTI
              # BP_NMII & TMS_INTI & !DSP_INTI
              # INTA4I & BP_NMII & TMS_INTI
              # INTA2I & BP_NMII & TMS_INTI
              # INTA0I & BP_NMII & TMS_INTI;
BP_IPL2I    = !INTA0I & !INTA2I & !INTA4I & BP_NMII & TMS_INTI & DSP_INTI;
END;
END;

SYM GLB B0 1 MPIPLS;
SIGTYPE MP_IPL0I OUT;
SIGTYPE MP_IPL1I OUT;
SIGTYPE MP_IPL2I OUT;
EQUATIONS
MP_IPL0I    = !INTAMP0I & !INTAMP2I & !INTAMP4I & MP_NMII & !ROLL_TICKI
              # !INTAMP0I & !INTAMP2I & !INTAMP4I & OS_TICKI & MP_NMII
              # MP_NMII & EXP_TICKI;
MP_IPL1I    = !INTAMP1I & !INTAMP3I & !INTAMP5I & OS_TICKI & MP_NMII & !EXP_TICKI
              # MP_NMII & !EXP_TICKI & ROLL_TICKI
              # INTAMP4I & MP_NMII & !EXP_TICKI
              # INTAMP2I & MP_NMII & !EXP_TICKI
              # INTAMP0I & MP_NMII & !EXP_TICKI;
MP_IPL2I    = !INTAMP0I & !INTAMP2I & !INTAMP4I & MP_NMII & !EXP_TICKI & !ROLL_TICKI;
END;
END;

```


ispLSI and pLSI: A Multiple Function Solution

```
// 16 BIT COUNTERS
SYM GLB D0 1 Q03; // BITS Q0-Q3
SIGTYPE [QQ_0..QQ_3] REG OUT; // SIGNIFIES A REGISTERED OUTPUT
EQUATIONS
QQ_0.CLK = CLK;
QQ_1.CLK = CLK;
QQ_2.CLK = CLK;
QQ_3.CLK = CLK;
QQ_0 = QQ_0 $$ VCC ;
QQ_1 = QQ_1 $$ QQ_0;
QQ_2 = QQ_2 $$ QQ_1 & QQ_0 ;
QQ_3 = QQ_3 $$ QQ_2 & QQ_1 & QQ_0 ;
END;
END;

SYM GLB D1 1 Q47; // BITS Q4-Q7
SIGTYPE [QQ_4..QQ_7] REG OUT;
EQUATIONS
QQ_4.CLK = CLK;
QQ_5.CLK = CLK;
QQ_6.CLK = CLK;
QQ_7.CLK = CLK;
QQ_4 = QQ_4 $$ QQ_3 & QQ_2 & QQ_1 & QQ_0 ;
QQ_5 = QQ_5 $$ QQ_4 & QQ_3 & QQ_2 & QQ_1 & QQ_0 ;
QQ_6 = QQ_6 $$ QQ_5 & QQ_4 & QQ_3 & QQ_2 & QQ_1 & QQ_0 ;
QQ_7 = QQ_7 $$ QQ_6 & QQ_5 & QQ_4 & QQ_3 & QQ_2 & QQ_1 & QQ_0 ;
END;
END;

SYM GLB D2 1 Q811; // BITS Q8-Q11
SIGTYPE [QQ_8..QQ_11] REG OUT;
EQUATIONS
QQ_8.CLK = CLK;
QQ_9.CLK = CLK;
QQ_10.CLK = CLK;
QQ_11.CLK = CLK;
QQ_8 = QQ_8 $$ TC_1 ;
QQ_9 = QQ_9 $$ QQ_8 & TC_1 ;
QQ_10 = QQ_10 $$ QQ_9 & QQ_8 & TC_1 ;
QQ_11 = QQ_11 $$ QQ_10 & QQ_9 & QQ_8 & TC_1 ;
END;
END;

SYM GLB D3 1 Q1215; // BITS Q12-Q15
SIGTYPE [QQ_12..QQ_15] REG OUT;
EQUATIONS
QQ_12.CLK = CLK;
QQ_13.CLK = CLK;
QQ_14.CLK = CLK;
QQ_15.CLK = CLK;
QQ_12 = QQ_12 $$ QQ_11 & QQ_10 & QQ_9 & QQ_8 & TC_1 ;
QQ_13 = QQ_13 $$ QQ_12 & QQ_11 & QQ_10 & QQ_9 & QQ_8 & TC_1 ;
QQ_14 = QQ_14 $$ QQ_13 & QQ_12 & QQ_11 & QQ_10 & QQ_9 & QQ_8 & TC_1 ;
QQ_15 = QQ_15 $$ QQ_14 & QQ_13 & QQ_12 & QQ_11 & QQ_10 & QQ_9 & QQ_8 & TC_1 ;
```

ispLSI and pLSI: A Multiple Function Solution

```

QQ_15 = QQ_15 $$ QQ_14 & QQ_13 & QQ_12 & QQ_11 & QQ_10 & QQ_9 & QQ_8 & TC_1 ;
END;
END;

SYM GLB D4 1 TC1;          // CARRY OUT OF BITS Q0-Q7 = HIGH
SIGTYPE TC_1 OUT;
EQUATIONS
TC_1 = (QQ_0 & QQ_1 & QQ_2 & QQ_3 & QQ_4 & QQ_5 & QQ_6 & QQ_7);
END;
END;

SYM GLB C3 1 TC2;          // CARRY OUT OF BITS Q0-Q15 = HIGH
SIGTYPE TC_2 OUT;
EQUATIONS
TC_2 = (QQ_8 & QQ_9 & QQ_10 & QQ_11 & QQ_12 & QQ_13 & QQ_14 & QQ_15 & TC_1);
END;
END;

SYM GLB D5 1 TC3;          // CARRY OUT OF BITS Q0-Q23 = HIGH
SIGTYPE TC_3 OUT;
EQUATIONS
TC_3 = (QQ_16 & QQ_17 & QQ_18 & QQ_19 & QQ_20 & QQ_21 & QQ_22 & QQ_23 & TC_2);
END;
END;

SYM GLB D6 1 TERM;          // CARRY OUT OF BITS Q0-Q31 = HIGH
SIGTYPE TERMCNT REG OUT;
EQUATIONS
TERMCNT.PTCLK = (TC_1 & TC_2 & TC_3 & QQ_24 & QQ_25 & QQ_26 & QQ_27 & QQ_28 &
QQ_29 & QQ_30 & QQ_31);
TERMCNT = VCC;
END;
END;

SYM GLB D7 1 Q1619;          // BITS Q16-Q19
SIGTYPE [QQ_16..QQ_19] REG OUT;
EQUATIONS
QQ_16.CLK = CLK;
QQ_17.CLK = CLK;
QQ_18.CLK = CLK;
QQ_19.CLK = CLK;
QQ_16 = QQ_16 $$ TC_2 ;
QQ_17 = QQ_17 $$ QQ_16 & TC_2 ;
QQ_18 = QQ_18 $$ QQ_17 & QQ_16 & TC_2 ;
QQ_19 = QQ_19 $$ QQ_18 & QQ_17 & QQ_16 & TC_2 ;
END;
END;

```

ispLSI and pLSI: A Multiple Function Solution

```

SYM GLB C0 1 Q2023; // BITS Q20-Q23
SIGTYPE [QQ_20..QQ_23] REG OUT;
EQUATIONS
QQ_20.CLK = CLK;
QQ_21.CLK = CLK;
QQ_22.CLK = CLK;
QQ_23.CLK = CLK;
QQ_20 = QQ_20 $$ QQ_19 & QQ_18 & QQ_17 & QQ_16 & TC_2 ;
QQ_21 = QQ_21 $$ QQ_20 & QQ_19 & QQ_18 & QQ_17 & QQ_16 & TC_2 ;
QQ_22 = QQ_22 $$ QQ_21 & QQ_20 & QQ_19 & QQ_18 & QQ_17 & QQ_16 & TC_2 ;
QQ_23 = QQ_23 $$ QQ_22 & QQ_21 & QQ_20 & QQ_19 & QQ_18 & QQ_17 & QQ_16 & TC_2 ;
END;
END;

SYM GLB C1 1 Q2427; // BITS Q24-Q27
SIGTYPE [QQ_24..QQ_27] REG OUT;
EQUATIONS
QQ_24.CLK = CLK;
QQ_25.CLK = CLK;
QQ_26.CLK = CLK;
QQ_27.CLK = CLK;
QQ_24 = QQ_24 $$ TC_3 ;
QQ_25 = QQ_25 $$ QQ_24 & TC_3 ;
QQ_26 = QQ_26 $$ QQ_25 & QQ_24 & TC_3 ;
QQ_27 = QQ_27 $$ QQ_26 & QQ_25 & QQ_24 & TC_3 ;
END;
END;

SYM GLB C2 1 Q2831; // BITS Q28-Q31
SIGTYPE [QQ_28..QQ_31] REG OUT;
EQUATIONS
QQ_28.CLK = CLK;
QQ_29.CLK = CLK;
QQ_30.CLK = CLK;
QQ_31.CLK = CLK;
QQ_28 = QQ_28 $$ QQ_27 & QQ_26 & QQ_25 & QQ_24 & TC_3 ;
QQ_29 = QQ_29 $$ QQ_28 & QQ_27 & QQ_26 & QQ_25 & QQ_24 & TC_3 ;
QQ_30 = QQ_30 $$ QQ_29 & QQ_28 & QQ_27 & QQ_26 & QQ_25 & QQ_24 & TC_3 ;
QQ_31 = QQ_31 $$ QQ_30 & QQ_29 & QQ_28 & QQ_27 & QQ_26 & QQ_25 & QQ_24 & TC_3 ;
END;
END;

// MULTIPLEXER GLBs
// SELECT HI ORDER BITS (16-31) IF !CNTELO
// OR SELECT LOW ORDER BITS (0-15) IF CNTELO
SYM GLB B1 1 MDAT01;
SIGTYPE OMDATA0I REG OUT;
SIGTYPE OMDATA1I REG OUT;
EQUATIONS
XCNT_SEL1.OE = XCNT_SEL1;
OMDATA0I.CLK = CNT_LTCH;
OMDATA0I = (!CNTELO & QQ_16 ) # (CNTELO & QQ_0 );
OMDATA1I = (!CNTELO & QQ_17 ) # (CNTELO & QQ_1 );
END;
END;

```

ispLSI and pLSI: A Multiple Function Solution

```

SYM GLB C4 1 MDATA23;
SIGTYPE OMDATA2I REG OUT;
SIGTYPE OMDATA3I REG OUT;
EQUATIONS
OMDATA2I.CLK = CNT_LTCH;
OMDATA2I = (!CNTELO & QQ_18 ) # (CNTELO & QQ_2 );
OMDATA3I = (!CNTELO & QQ_19 ) # (CNTELO & QQ_3 );
END;
END;

SYM GLB B2 1 MDATA45;
SIGTYPE OMDATA4I REG OUT;
SIGTYPE OMDATA5I REG OUT;
EQUATIONS
OMDATA4I.CLK = CNT_LTCH;
OMDATA4I = (!CNTELO & QQ_20 ) # (CNTELO & QQ_4 );
OMDATA5I = (!CNTELO & QQ_21 ) # (CNTELO & QQ_5 );
END;
END;

SYM GLB C5 1 MDATA67;
SIGTYPE OMDATA6I REG OUT;
SIGTYPE OMDATA7I REG OUT;
EQUATIONS
OMDATA6I.CLK = CNT_LTCH;
OMDATA6I = (!CNTELO & QQ_22 ) # (CNTELO & QQ_6 );
OMDATA7I = (!CNTELO & QQ_23 ) # (CNTELO & QQ_7 );
END;
END;

SYM GLB C6 1 MDATA811;
SIGTYPE OMDATA8I REG OUT;
SIGTYPE OMDATA9I REG OUT;
SIGTYPE OMDATA10I REG OUT;
SIGTYPE OMDATA11I REG OUT;
EQUATIONS
XCNT_SEL.OE = XCNT_SEL;
OMDATA8I.CLK = CNT_LTCH;
OMDATA8I = (!CNTELO & QQ_24 ) # (CNTELO & QQ_8 );
OMDATA9I = (!CNTELO & QQ_25 ) # (CNTELO & QQ_9 );
OMDATA10I = (!CNTELO & QQ_26 ) # (CNTELO & QQ_10 );
OMDATA11I = (!CNTELO & QQ_27 ) # (CNTELO & QQ_11 );
END;
END;

```

ispLSI and pLSI: A Multiple Function Solution

```

SYM GLB C7 1 MDAT1215;
SIGTYPE OMDATA12I REG OUT;
SIGTYPE OMDATA13I REG OUT;
SIGTYPE OMDATA14I REG OUT;
SIGTYPE OMDATA15I REG OUT;
EQUATIONS
OMDATA12I.CLK = CNT_LTCH;
OMDATA12I = (!CNTELO & QQ_28) # (CNTELO & QQ_12);
OMDATA13I = (!CNTELO & QQ_29) # (CNTELO & QQ_13);
OMDATA14I = (!CNTELO & QQ_30) # (CNTELO & QQ_14);
OMDATA15I = (!CNTELO & QQ_31) # (CNTELO & QQ_15);
END;
END;

SYM GLB A1 1 IOMEMOE;
SIGTYPE IO_SELECT0 OUT CRIT;
SIGTYPE IO_SELECT1 OUT CRIT; // Signifies ORP Bypass
SIGTYPE MEMOE OUT;
EQUATIONS
IO_SELECT0 = MPA23 & MPA22 & !MPA21 & MPIO_MEM & !MPWR_RD;
IO_SELECT1 = MPA23 & !MPA22 & MPA21 & MPIO_MEM & MPWR_RD;
MEMOE = !MPIO_MEM & !MPWR_RD & MPRDY;
END;
END;

SYM GLB A0 1 MEMCSWR;
SIGTYPE MEMCS REG OUT;
SIGTYPE MEMWR REG OUT;
EQUATIONS
MEMCS.CLK = CLK;
MEMCS = MPA23 & !MPA22 & !MPA21 & !MPIO_MEM # MEMCS & MPRDY; // CHIP
SELECT
MEMWR = MPA23 & !MPA22 & !MPA21 & !MPIO_MEM & MPWR_RD # MEMWR & MPRDY; // MEMORY
WRITE
OR READ
END;
END;

// IO CELL ASSIGNMENTS
SYM IOC IO51 1 MPIPLS;
XPIN IO DSP_INT;
IB11 (DSP_INTI,DSP_INT);
END;

SYM IOC IO0 1 MPDAT15;
XPIN IO MP_DATA15 ;
BI11 (MP_DATA15I,MP_DATA15,INTAMP5I,MP_INT_RDI);
END;

SYM IOC IO1 1 MPDAT14;
XPIN IO MP_DATA14;
BI11 (MP_DATA14I,MP_DATA14,INTAMP4I,MP_INT_RDI); //BI11 = BIDIRECTIONAL I/O
END;

```


ispLSI and pLSI: A Multiple Function Solution

```
SYM IOC IO2 1 MPDAT13;
XPIN IO MP_DATA13;
BI11 (MP_DATA13I,MP_DATA13,INTAMP3I,MP_INT_RDI);
END;
```

```
SYM IOC IO3 1 MPDAT12;
XPIN IO MP_DATA12;
BI11 (MP_DATA12I,MP_DATA12,INTAMP2I,MP_INT_RDI);
END;
```

```
SYM IOC IO4 1 MPDAT11;
XPIN IO MP_DATA11;
BI11 (MP_DATA11I,MP_DATA11,INTAMP1I,MP_INT_RDI);
END;
```

```
SYM IOC IO5 1 MPDAT10;
XPIN IO MP_DATA10;
BI11 (MP_DATA10I,MP_DATA10,INTAMP0I,MP_INT_RDI);
END;
```

```
SYM IOC IO8 1 MPINTCL;
XPIN IO MP_INT_CLR LOCK 40;
IB11 (MP_INT_CLRI,MP_INT_CLR);
END; // LOCK = FIXED PIN
```

```
SYM IOC IO9 1 RSET;
XPIN IO RSET;
IB11 (RSETI,RSET);
END;
```

```
SYM IOC IO10 1 MDATA15;
XPIN IO MDATA15 LOCK 53 ;
OT11 (MDATA15,OMDATA15I,!XCNT_SEL);
END; // TRISTATE OUTPUT
```

```
SYM IOC IO11 1 MDATA14;
XPIN IO MDATA14 LOCK 54 ;
OT11 (MDATA14,OMDATA14I,!XCNT_SEL);
END;
```

```
SYM IOC IO12 1 MDATA13;
XPIN IO MDATA13 LOCK 55 ;
OT11 (MDATA13,OMDATA13I,!XCNT_SEL);
END;
```

```
SYM IOC IO13 1 MDATA12;
XPIN IO MDATA12 LOCK 56 ;
BI11 (MDATA12I,MDATA12,OMDATA12I,!XCNT_SEL);
END;
```

ispLSI and pLSI: A Multiple Function Solution

```
SYM IOC IO14 1 MDATA11;  
XPIN IO MDATA11 LOCK 57 ;  
BI11 (MDATA11I,MDATA11,OMDATA11I,!XCNT_SEL);  
END;
```

```
SYM IOC IO15 1 MDATA10;  
XPIN IO MDATA10 LOCK 58 ;  
BI11 (MDATA10I,MDATA10,OMDATA10I,!XCNT_SEL);  
END;
```

```
SYM IOC IO16 1 MDATA9;  
XPIN IO MDATA9 LOCK 59 ;  
OT11 (MDATA9,OMDATA9I,!XCNT_SEL);  
END;
```

```
SYM IOC IO17 1 MDATA8;  
XPIN IO MDATA8 LOCK 60 ;  
BI11 (MDATA8I,MDATA8,OMDATA8I,!XCNT_SEL);  
END;
```

```
SYM IOC IO26 1 MAS;  
XPIN IO MASX LOCK 27 ;  
IB11 (MAS,MASX);  
END;
```

```
SYM IOC IO27 1 IPC_INT;  
XPIN IO IPC_INT LOCK 26 ;  
IB11 (IPC_INTI,IPC_INT);  
END;
```

```
SYM IOC IO28 1 MPIPL2;  
XPIN IO MP_IPL2;  
OB11 (MP_IPL2,MP_IPL2I);  
END;
```

```
SYM IOC IO29 1 MPIPL1;  
XPIN IO MP_IPL1;  
OB11 (MP_IPL1,MP_IPL1I);  
END;
```

```
SYM IOC IO30 1 MPIPL0;  
XPIN IO MP_IPL0;  
OB11 (MP_IPL0,MP_IPL0I);  
END;
```

```
SYM IOC IO31 1 MPINTRD;  
XPIN IO MP_INT_RD;  
IB11 (MP_INT_RDO,MP_INT_RD);  
END;
```

ispLSI and pLSI: A Multiple Function Solution

```

SYM IOC IO32 1 BPINTRD;
XPIN IO BP_INT_RD;
IB11 (BP_INT_RDO,BP_INT_RD);
END;

SYM IOC IO33 1 BPDAT15;
XPIN IO BP_DATA15;
BI11 (BP_DATA15I,BP_DATA15,INTA5I,BP_INT_RDI);
END;

SYM IOC IO34 1 BPDAT14;
XPIN IO BP_DATA14;
BI11 (BP_DATA14I,BP_DATA14,INTA4I,BP_INT_RDI);
END;

SYM IOC IO35 1 BPDAT13;
XPIN IO BP_DATA13;
BI11 (BP_DATA13I,BP_DATA13,INTA3I,BP_INT_RDI);
END;

SYM IOC IO36 1 BPDAT12;
XPIN IO BP_DATA12;
BI11 (BP_DATA12I,BP_DATA12,INTA2I,BP_INT_RDI);
END;

SYM IOC IO37 1 BPDAT11;
XPIN IO BP_DATA11;
BI11 (BP_DATA11I,BP_DATA11,INTA1I,BP_INT_RDI);
END;

SYM IOC IO38 1 BPDAT10;
XPIN IO BP_DATA10;
BI11 (BP_DATA10I,BP_DATA10,INTA0I,BP_INT_RDI);
END;

SYM IOC IO41 1 BPINTCL;
XPIN IO BP_INT_CLR;
IB11 (BP_INT_CLRI,BP_INT_CLR);
END;

SYM IOC IO42 1 BPIPL2;
XPIN IO BP_IPL2;
OB11 (BP_IPL2,BP_IPL2I);
END;

SYM IOC IO43 1 BPIPL1;
XPIN IO BP_IPL1;
OB11 (BP_IPL1,BP_IPL1I);
END;

SYM IOC IO44 1 BPIPL0;
XPIN IO BP_IPL0;
OB11 (BP_IPL0,BP_IPL0I);
END;

```

ispLSI and pLSI: A Multiple Function Solution

```
SYM IOC IO45 1 MP_NMI;  
XPIN IO MP_NMI;  
IB11 (MP_NMI1,MP_NMI);  
END;
```

```
SYM IOC IO46 1 OS_TICK;  
XPIN IO OS_TICK;  
IB11 (OS_TICK1,OS_TICK);  
END;
```

```
SYM IOC IO47 1 EXPTICK;  
XPIN IO EXP_TICK;  
IB11 (EXP_TICK1,EXP_TICK);  
END;
```

```
SYM IOC IO48 1 ROLTICK;  
XPIN IO ROLL_TICK;  
IB11 (ROLL_TICK1,ROLL_TICK);  
END;
```

```
SYM IOC IO49 1 BP_NMI;  
XPIN IO BP_NMI;  
IB11 (BP_NMI1,BP_NMI);  
END;
```

```
SYM IOC IO50 1 TMS_INT;  
XPIN IO TMS_INT;  
IB11 (TMS_INT1,TMS_INT);  
END;
```

```
SYM IOC IO18 1 MPCLR13;  
XPIN IO MDATA7;  
OT11 (MDATA7,OMDATA7I,!XCNT_SEL1);  
END;
```

```
SYM IOC IO19 1 MPCLR13;  
XPIN IO MDATA6;  
OT11 (MDATA6,OMDATA6I,!XCNT_SEL1);  
END;
```

```
SYM IOC IO20 1 MPCLR13;  
XPIN IO MDATA5 LOCK 6;  
OT11 (MDATA5,OMDATA5I,!XCNT_SEL1);  
END;
```

```
SYM IOC IO21 1 MPCLR13;  
XPIN IO MDATA4 LOCK 5;  
OT11 (MDATA4,OMDATA4I,!XCNT_SEL1);  
END;
```

```
SYM IOC IO22 1 MPCLR13;  
XPIN IO MDATA3;  
OT11 (MDATA3,OMDATA3I,!XCNT_SEL1);  
END;
```

ispLSI and pLSI: A Multiple Function Solution

```
SYM IOC IO23 1 MPCLR13;
XPIN IO MDATA2;
OT11 (MDATA2,OMDATA2I,!XCNT_SEL1);
END;
```

```
SYM IOC IO24 1 MPCLR13;
XPIN IO MDATA1 LOCK 4 ;
OT11 (MDATA1,OMDATA1I,!XCNT_SEL1);
END;
```

```
SYM IOC IO25 1 MPCLR13;
XPIN IO MDATA0 LOCK 3 ;
OT11 (MDATA0,OMDATA0I,!XCNT_SEL1);
END;
```

```
SYM IOC IO63 1 LTCH;
XPIN IO XCNTSEL;
IB11 (XCNT_SEL1, XCNTSEL);
END;
```

```
SYM IOC Y1 1 LTCH;
XPIN CLK LTCH;
IB11 (CNT_LTCH,LTCH);
END;
```

```
SYM IOC Y0 1 CLOCK;
XPIN CLK XCLK;
IB11 (CLK, XCLK);
END;
```

```
SYM IOC IO62 1 CNTELO;
XPIN IO OCNTELO;
IB11 (CNTELO, OCNTELO);
END;
```

```
SYM IOC IO61 1 TERMCNT;
XPIN IO XTERMCNT;
OB11 (XTERMCNT, TERMCNT);
END;
```

```
SYM IOC IO59 1 MPA23;
XPIN IO MPA230;
IB11 (MPA23, MPA230);
END;
```

```
SYM IOC IO58 1 MPA21;
XPIN IO MPA220;
IB11 (MPA22, MPA220);
END;
```

```
SYM IOC IO57 1 MPA21;
XPIN IO MPA210;
IB11 (MPA21, MPA210);
END;
```

```
SYM IOC IO23 1 MPCLR13;
XPIN IO MDATA2;
OT11 (MDATA2,OMDATA2I,!XCNT_SEL1);
END;
```

```
SYM IOC IO24 1 MPCLR13;
XPIN IO MDATA1 LOCK 4 ;
OT11 (MDATA1,OMDATA1I,!XCNT_SEL1);
END;
```

```
SYM IOC IO25 1 MPCLR13;
XPIN IO MDATA0 LOCK 3 ;
OT11 (MDATA0,OMDATA0I,!XCNT_SEL1);
END;
```

```
SYM IOC IO63 1 LTCH;
XPIN IO XCNTSEL;
IB11 (XCNT_SEL1, XCNTSEL);
END;
```

```
SYM IOC Y1 1 LTCH;
XPIN CLK LTCH;
IB11 (CNT_LTCH,LTCH);
END;
```

```
SYM IOC Y0 1 CLOCK;
XPIN CLK XCLK;
IB11 (CLK, XCLK);
END;
```

```
SYM IOC IO62 1 CNTELO;
XPIN IO OCNTELO;
IB11 (CNTELO, OCNTELO);
END;
```

```
END; //LDS DESIGNER
```


isPLSI and pLSI: A Multiple Function Solution

```

SYM IOC IO56 1 MPIO_MEM;
XPIN IO MPIO_MEMO;
IB11 (MPIO_MEM, MPIO_MEMO);
END;

SYM IOC IO55 1 MPWR_RD;
XPIN IO MPWR_RDO;
ID11 (MPWR_RD, MPWR_RDO,CLK);
END;

SYM IOC IO54 1 MPRDY;
XPIN IO MP_RDYO;
IB11 (MPRDY, MP_RDYO);
END;

SYM IOC IO53 1 IO_SELECT;
XPIN IO IO_SELECT00;
OB11 (IO_SELECT00, IO_SELECT0);
END;

SYM IOC IO52 1 IO_SELECT1;
XPIN IO IO_SELECT10;
OB11 (IO_SELECT10, IO_SELECT1);
END;

SYM IOC IO7 1 MEMCS;
XPIN IO MEMCSO ;
OB11 (MEMCSO, MEMCS);
END;

SYM IOC IO60 1 MEMOE;
XPIN IO MEMOEO;
OB11 (MEMOEO, MEMOE);
END;
//LDF DESIGNLDF

```

Introduction

There are several ways to program multiple In-System Programmable (ISP™) devices — each ISP device can be programmed individually through an independent ISP interface, or multiple devices can share a parallel multiplexed or serial daisy chained interface. Each method has its unique advantages. The serial daisy chain method is the most efficient and easiest to implement as it uses a simple hardware interface and programming procedures.

This applications note explains how to program multiple ISP devices in a daisy chained configuration. It will also explain the general ISP programming interface and the unique programming features of each ISP device.

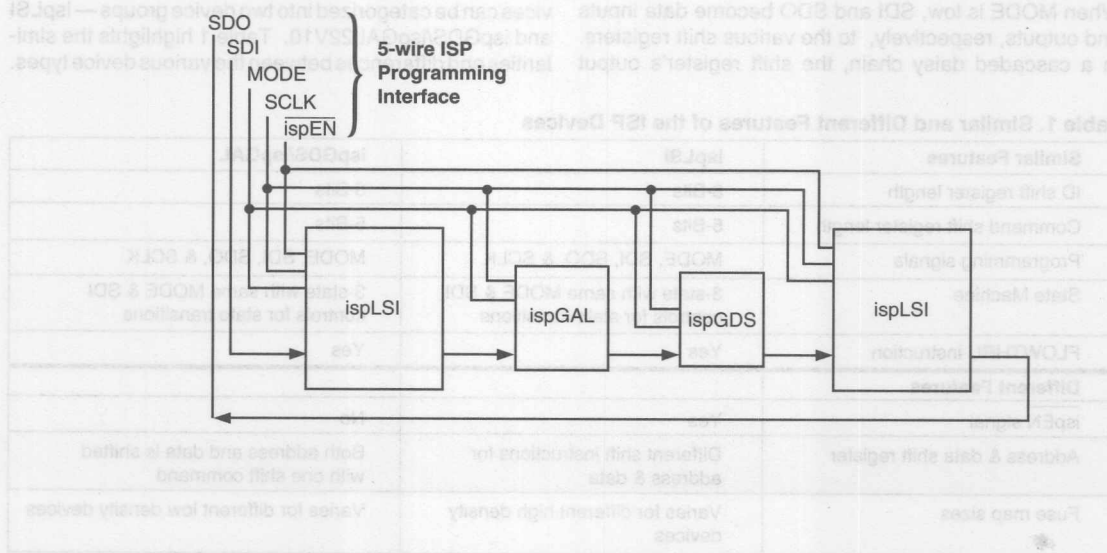
ISP Overview

Programming Interface

Programming of Lattice's ispLSI™, ispGAL®, and ispGDS™ devices is based on a similar programming

interface. The basic components of the ISP programming interface are the three-state programming control state machine and mode control (MODE), serial data in (SDI), serial data out (SDO) and serial clock (SCLK) inputs. The state machine built into each ISP device is controlled by three inputs — MODE, SDI and SCLK. In addition, ispLSI devices use a fourth input, ispEN, to multiplex the functions of the SDI, SDO, SCLK and MODE pins between the ISP programming functions and user defined logic signals during normal PLD operation. The state machine controls the sequence of programming operations such as identifying the ISP device, shifting in appropriate data and commands, programming pulse widths, and erasing the device. All programming information is shifted in and out of the device serially through the SDI and SDO pins. Each ISP device comes with a unique eight bit hardwired device ID to make the electronic identification of the devices by the programming software easy. The following sections explain the ISP programming interface using multiple daisy chained ISP devices. Figure 1 illustrates a typical block diagram of multiple ISP devices cascaded together.

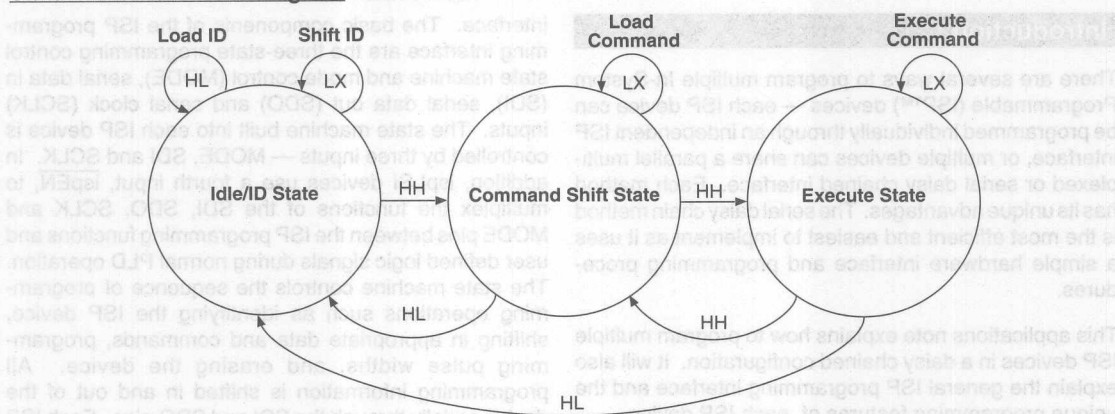
Figure 1. Multiple ISP Device Programming Interface



Programming Multiple ISP Devices

Figure 2. ISP State Machine

State Transition Control Signals : MODE SDI



State Machine

The state transitions of the three-state state machine shown in figure 2 are controlled by the MODE and SDI signals. Within each state the MODE signal directs whether SDI is a control input (MODE = H) or SDI is a data input (MODE = L). When MODE is high, the SDI's logic level is reflected on SDO. This feature allows devices to transparently pass the SDI control input to devices further down the daisy chain.

When MODE is low, SDI and SDO become data inputs and outputs, respectively, to the various shift registers. In a cascaded daisy chain, the shift register's output

(SDO) is connected to the next device's shift register input (SDI). Programming data is shifted into the SDI input of the first device in the daisy chain. All shift registers in the daisy chained devices are connected together so data can be shifted to the last device's SDO where the ISP programming controller can verify the data.

Similarities and Differences Between Devices

For the purpose of cascading the ISP devices, the devices can be categorized into two device groups — ispLSI and ispGDS/ispGAL22V10. Table 1 highlights the similarities and differences between the various device types.

Table 1. Similar and Different Features of the ISP Devices

Similar Features	ispLSI	ispGDS/ispGAL
ID shift register length	8-Bits	8-Bits
Command shift register length	5-Bits	5-Bits
Programming signals	MODE, SDI, SDO, & SCLK	MODE, SDI, SDO, & SCLK
State Machine	3-state with same MODE & SDI controls for state transitions	3-state with same MODE & SDI controls for state transitions
FLOWTHRU instruction	Yes	Yes
Different Features		
ispEN signal	Yes	No
Address & data shift register	Different shift instructions for address & data	Both address and data is shifted with one shift command
Fuse map sizes	Varies for different high density devices	Varies for different low density devices

Programming Multiple ISP Devices

Using the same state machine controls makes it possible to program multiple ISP devices by operating all the cascaded devices' state machines in parallel. This synchronizes all the devices during programming within the daisy chain to a known state. However, having all ISP devices in the same state does not mean that all devices are executing the same instruction. The ability of each device in the daisy chain to execute a different instruction makes selectively programming one or multiple ISP devices at a time possible.

For the ispLSI devices, the active `ispEN` signal enables the programming mode of the device. By driving `ispEN` low, all I/Os of the devices are put into a high-impedance state for programming and the programming functions for SDI, SDO, Mode and SCLK are enabled. A difference in the ispGDS/ispGAL devices is that the I/Os are put into a high-impedance state when the programming state machine goes into Command Shift State. The ispGDS/ispGAL devices do not use a dedicated `ispEN` pin for this function.

Most shift operations such as ID shift and command shift operations are the same between the ispLSI and the ispGDS/ispGAL devices. One shift operation that is different between the two types of devices is the way the address and data is shifted into the devices. The ispLSI devices have separate address and data shift commands. The row(s) are selected by the address that is shifted-in prior to each programming command for that row. The data can then be shifted with the data shift instruction. In the case of ispGDS/ispGAL devices, both address and data are shifted-in with a single shift com-

mand (the address is part of the data shift register). When executing commands that only require a row address, a dummy data stream or no data can be shifted in place of the data stream.

With an understanding of the ISP programming interface and the differences between different types of devices, a specific daisy chained design example will be used to illustrate the details of programming different ISP devices.

Daisy Chained Interface

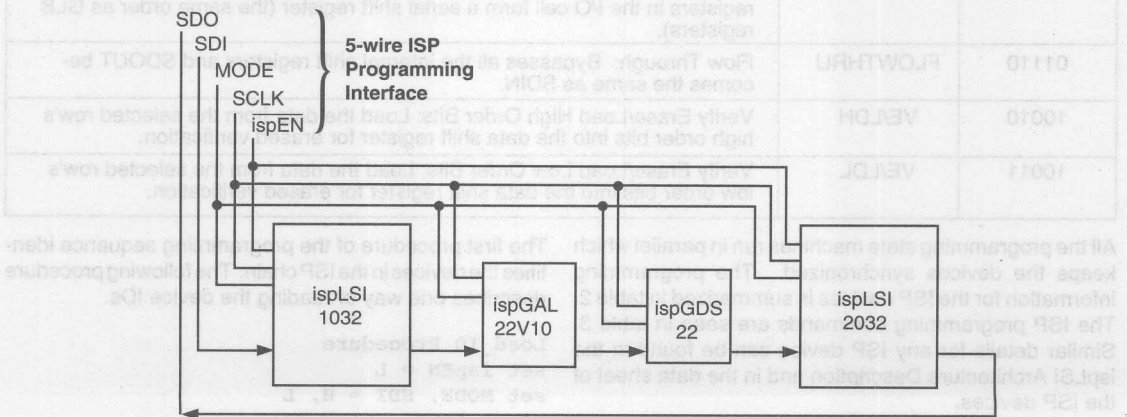
Advantages

One of the main advantages of daisy chained ISP programming is the simplified hardware interface. The number of ISP devices that can be connected to the same 5-wire interface is limited only by the signal drive capability of the ISP programming control logic. One serial daisy chain is capable of providing all the necessary programming interface which minimizes the hardware overhead for in-system programming. Software controls generated from PCs, microcontrollers and test equipments can program and reconfigure all ISP devices during various board level design, test, and manufacturing stages.

ISP Daisy Chain Programming

A specific illustration of multiple device programming in a daisy chained environment is shown in figure 3. The example shows the ISP programming aspects such as identifying the devices in the daisy chain, shifting commands, bypassing devices, and executing commands.

Figure 3. ISP Daisy Chain Example



Programming Multiple ISP Devices

Table 2. ISP Programming Information

Description	ispLSI 1032	ispGAL22V10	ispGDS22	ispLSI 2032
Device ID (8-bits)	0000 0011	0000 1000	0111 0010	0001 0101
Command Register	5 bits	5 bits	5 bits	5 bits
Address Shift Register	108 bits	n/a	n/a	102 bits
Data/Addr. & Data Shift Register	160 bits	(6+132) bits	(6+18) bits	40 bits

Table 3. State Machine Instruction Set

Instruction	Operation	Description
00000	NOP	No operation performed
00001	ADDSHFT	Address Register Shift: Shifts address into the address shift register from SDIN.
00010	DATASHFT	Data Register Shift: Shifts data into or out of the data serial shift register.
00011	UBE	User Bulk Erase: Erase the entire device.
00100	GRPBE	Global Routing Pool Bulk Erase: Bulk erases the GRP array only.
00101	GLBBE	Generic Logic Block Bulk Erase: Bulk erases all the GLB array only.
00110	ARCHBE	Architecture Bulk Erase: Bulk erases the architecture array and I/O configuration only.
00111	PRGMH	Program High Order Bits: The data in the data shift register is programmed into the addressed row's high order bits.
01000	PRGML	Program Low Order Bits: The data in the data shift register is programmed into the addressed row's low order bits.
01001	PRGMS	Program Security Cell: Programs the security cell of the device.
01010	VER/LDH	Verify/Load High Order Bits: Load the data from the selected row's high order bits into the data shift register for verification.
01011	VER/LDL	Verify/Load Low Order Bits: Load the data from the selected row's low order bits into the data shift register for verification.
01100	GLBPRLD	Generic Logic Block Preload: Preloads the registers in the GLB with the data from SDIN. All registers in the GLB form a serial shift register. Refer to device layout section for details.
01101	IOPRLD	I/O Preload: Preloads the I/O registers with the data from SDIN. All registers in the I/O cell form a serial shift register (the same order as GLB registers).
01110	FLOWTHRU	Flow Through: Bypasses all the internal shift registers and SDOUT becomes the same as SDIN.
10010	VE/LDH	Verify Erase/Load High Order Bits: Load the data from the selected row's high order bits into the data shift register for erased verification.
10011	VE/LDL	Verify Erase/Load Low Order Bits: Load the data from the selected row's low order bits into the data shift register for erased verification.

All the programming state machines run in parallel which keeps the devices synchronized. The programming information for the ISP devices is summarized in table 2. The ISP programming commands are seen in table 3. Similar details for any ISP device can be found in the ispLSI Architecture Description and in the data sheet of the ISP devices.

The first procedure of the programming sequence identifies the devices in the ISP chain. The following procedure describes one way of reading the device IDs.

Load_ID Procedure

```

set ispEN = L
set MODE, SDI = H, L
clock SCLK (Load ID)
Continue to Shift_ID Procedure ...

```


Programming Multiple ISP Devices

At this point the 8-bit ID registers are loaded with the hardwired device IDs. Figure 4 shows the configuration of the ID shift registers.

After the device ID has been loaded, the following shift ID procedure sequentially shifts the IDs through to the last device's SDO. While the ID is being shifted out, keep SDI at a known logic level so that the end of the ID stream can be identified. This is especially important when an unknown number of devices are in the ISP daisy chain. By detecting a sequence of 8 zeros or 8 ones, the ISP controller can detect the end of the ID string.

Shift_ID Procedure

... Continued from Load_ID Procedure

```
set MODE, SDI = L, H
```

```
clock SCLK (Shift ID)
```

```
if last 8 SDO = H then goto End
```

```
else goto Shift_ID
```

```
End
```

At this point all devices within the ISP daisy chain and their order in the chain can be properly identified. The next step is to match the proper JEDEC fuse map file to the appropriate device. There are several programming options at this point. To simplify the programming routines, this example programs the devices one at a time.

The following procedures illustrate how to shift commands, shift data and execute the commands to program the ispGAL22V10. Since the 22V10 is the second device of the ISP daisy chain, these procedures also illustrate how to put the other devices into flow through mode. The

following procedure shifts SHIFT_DATA command into the 22V10 and FLOWTHRU command into the rest of the ISP devices.

Load_Command Procedure

... Continued from end of Shift_ID Procedure

```
set MODE, SDI = H, H
```

```
clock SCLK (Shift State)
```

```
set MODE = L
```

```
Loop
```

```
set SDI = command stream (figure 5)
```

```
clock SCLK (Shift Command)
```

```
End Loop
```

```
End Procedure
```

Execute_Command Procedure

```
set MODE, SDI = H, H
```

```
clock SCLK (Execute State)
```

```
set MODE = L
```

```
Loop 138 times
```

```
set SDI = data stream (figure 6)
```

```
clock SCLK (Execute
```

```
SHIFT_DATA Command)
```

```
End Loop
```

```
set MODE, SDI = H, H
```

```
clock SCLK (Shift State)
```

```
End Procedure
```

Figure 4. ID Shift Register Configuration

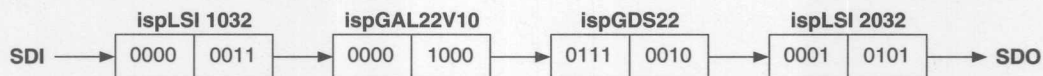


Figure 5. ISP Command Stream

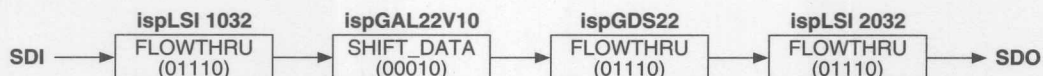
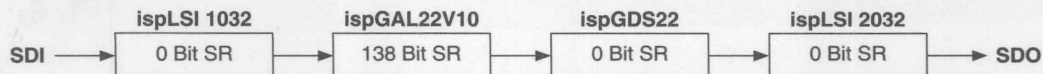


Figure 6. ISP Data Stream



At the end of the `Execute_Command` Procedure the state machine is returned to the Shift State. This readies the devices for another command shift procedure. For the `ispGAL22V10`, the `DATA_SHIFT` instruction of 138 bits includes the row address and the data associated with the row. Similar procedures can be used to complete the programming of the `ispGAL22V10`.

H/W & S/W Considerations

In order to keep the software procedures concise, Lattice recommends programming multiple devices of the same device type together. This creates modular shift routines and programming routines as opposed to having to make each routine a special case.

All `ispLSI` devices are shipped bulk erased which means all outputs are in high impedance state for the blank devices. The Lattice manufacturing outgoing pattern for

the `ispGDS/ispGAL22V10` devices also puts all I/Os in high impedance state. The `ispEN` signal controls whether the `ispLSI` device is in programming or normal mode. `MODE` and `SDI` controls this function on the `ispGDS/ispGAL` devices. It is recommended to put pull-down resistors on the `MODE` & `SDI` signals on the `ispGDS/ispGAL` devices in order to keep the default state of the device in normal operation mode.

Summary

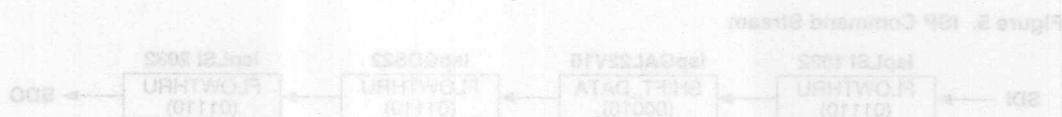
This applications note provides an example of one way to program multiple ISP devices. Daisy chaining provides an easy-to-implement, cost-effective means to program multiple ISP devices. The flexibility of ISP allows the user to customize the programming to fit a specific application. For additional information on ISP, contact your local Lattice sales representatives or call Lattice Literature and Applications support at 1-800-327-8425.

```

Execute_Command Procedure
    set MODE, SDI = H, H
    loop 138 times
        set SDI = data stream (Figure 6)
        clock SCLK (Execute)
    end loop
    set MODE, SDI = H, H
    clock SCLK (Shift State)
end procedure

```

At this point all devices within the ISP delay chain and the order in the chain can be properly identified. The next step is to match the proper JEDSO bus map file to the specific device. There are several programming options at this point. To simplify the programming routine, this example programs the devices one at a time. The following procedure illustrates how to shift commands and data and execute the commands to program the `ispGAL22V10`. Since the `22V10` is the second device in the ISP delay chain, these procedures also illustrate how to put the other devices into flow through mode. The





Compiling Multiple PLDs into ispLSI and pLSI Devices

Introduction

As high density programmable devices become more complex, they can combine larger designs previously implemented with low density PLDs and SSI/MSI glue logic. The use of ispLSI and pLSI devices from Lattice Semiconductor can reduce manufacturing costs by: shrinking board size, simplifying test procedures, speeding development, and reducing the type and number of parts required to be kept in inventory. Designers familiar with PLDs and SSI/MSI devices can convert to Lattice ispLSI and pLSI devices with little effort. This application note addresses a procedure to convert a circuit designed with PLDs, MSI, and SSI devices into the Lattice Semiconductor pLSI and ispLSI device format.

The basic steps required to convert the design are:

- ☐ Define the I/Os
- ☐ Convert the Low Density PLD Equations
- ☐ Combine the PLD Source Files
- ☐ Add any MSI, SSI Functions
- ☐ Partition the Logic into Generic Logic Blocks (GLBs)
- ☐ Import the File into the ispLSI and pLSI Design Environment
- ☐ Place and Route Using the pLSI and ispLSI Development System (pDS®)

Define the I/Os

The first task in the conversion process is to define the I/O pins of the Lattice ispLSI and pLSI device based on the circuit developed using lower density devices. One must determine if the design is I/O limited or gate limited. If the design is I/O limited the circuit must be partitioned into a higher pin count device, or two (or more) lower pin count devices. A gate limited design will mandate the design be partitioned into a higher density ispLSI and pLSI device. This implies that there will be unused I/O pins. This can allow additional functionality to be designed into the Lattice ispLSI and pLSI device, providing the device does not become gate limited again.

A straightforward approach to estimate gate count is to use SSI, MSI and PLD equivalents. By adding up the total number of these circuit blocks required for a circuit, one can determine if the design will fit into a Lattice ispLSI and pLSI device. For example, the 1000 and 2000 family GLB (Generic Logic Block) of the Lattice ispLSI and pLSI family has 18 inputs and 4 outputs. Numerous functions implemented in 16V8, 20V8 and 22V10 devices can be fit easily into one GLB. However, in cases where five or more outputs are desired, partitioning into 2 GLBs will be necessary. Expanding this analogy, approximately 1 MSI device and 2 SSI devices can fit into a single GLB.

When converting a circuit implemented with MSI, SSI and PLDs, partitioning can be achieved by recognizing which nodes are best suited for interconnection within the ispLSI and pLSI device. The partitioning of logic will vary for different MSI, SSI or PLD devices. By determining which of these devices will be implemented completely within the Lattice ispLSI and pLSI device, it will become readily apparent which of the nodes should be kept within the ispLSI and pLSI device or allocated as an I/O pin. Signals which connect to a device not implemented within the Lattice ispLSI and pLSI device will be required to be an I/O. As a shot gun approach, one can simply draw a box around the circuit, count the I/O and gate requirement, and select the ispLSI and pLSI device meeting the requisite gate and I/O count. This task requires good engineering judgement and knowledge of device architecture to effectively utilize the ispLSI and pLSI device architecture.

Nodes which have a broad fanout should be considered for I/O unless all destination devices are implemented within the Lattice ispLSI and pLSI device. Naturally, nodes going off-board must be implemented as I/O pins on the Lattice ispLSI and pLSI device.

Clocking is another factor to consider when partitioning a circuit. In the 1000 family, if the circuit requires more than the four global clocks available in the ispLSI and pLSI device, the circuit should be partitioned so that circuits with common clocks are in the same ispLSI and pLSI device. The global clock inputs are available on pins Y0, Y1, Y2 and Y3. Y0, Y1 and Y2 can be directly connected to any GLB, while Y2 and Y3 can be directly

4

$$\begin{aligned} \text{BSIG_C} &= \text{SIG_C} \\ \text{BSIG_C.OE} &= \text{OE_C} \end{aligned}$$

Listing 2. Multiplexer Equations

```

MUX_OUT = !OE_B & !OE_A & SIG_A #      // SELECT SIG_A
         !OE_B & OE_A & SIG_B #        // SELECT SIG_B
         OE_B & !OE_A & SIG_C;          // SELECT SIG_C

```

*Note that OE C is not needed in this implementation.

The AND function of the output enables (OE_A, OE_B) does not increase the number of product terms required to implement the various bus signal functions. This will always be true for product term oriented architectures such as the Lattice ispLSI and pLSI devices. There are ten ONE of N multiplexer Macros currently available in the ispLSI and pLSI Macro Library. By using these Macros, the conversion may be readily accomplished by simply changing the default signal names within the Lattice Macro.

Inversion Placement

Proper placement of active low internal signals may provide a significant savings in the utilization of the Lattice ispLSI and pLSI device resources as described in the following example.

Listing 3. Original Function Required

```
OUT = (!IN1 # !IN2 # !IN3) & (!IN4 # !IN5 # !IN6);
```

Listing 4. Showing Sum of Products Form of Listing 14

```
out = (!in3 & !in6
      # !in2 & !in6
      # !in1 & !in6
      # !in3 & !in5
      # !in2 & !in5
      # !in1 & !in5
      # !in3 & !in4
      # !in2 & !in4
      # !in1 & !in4):
```

Listing 5. Showing Reduction of Product Terms with ! Use

```
out = !((!in1 # !in2 # !in3) & (!in4 # !in5 # !in6));
out = (in1 & in2 & in3) # (in4 & in5 & in6);
```

```
// SELECT SIG_A
// SELECT SIG_B
// SELECT SIG_C
```

Consider the equations shown in Listing 3. The original was entered in a "Product of Sums" form which becomes the "Sum of Products" form shown in listing 4. Traditional PLDs require that logic be in Sum of Products form to be implemented in the architecture. This equation requires nine product terms to implement.

A better way to implement this function is to Demorganize (invert) the equation, as shown in listing 5. By doing this the implementation becomes two product terms versus nine for the non-inverted form. There are inversions available in each I/O cell and each input to the GLBs to re-invert the signal to get the original function.

Compiling Multiple PLDs into ispLSI and pLSI Devices

Therefore, when manipulating equations to fit the Lattice ispLSI and pLSI architecture, consider placing inversions for active low outputs at the signal destination or at the I/O cell. The Lattice ispLSI and pLSI family can accommodate any active low signal with this technique as all inputs to the logic block have both true and complementary inputs. In other words a signal "A" routed to a GLB, will have both "A" and "IA" available within the GLB AND array. The outputs of the Lattice ispLSI and pLSI devices can also be selected as active high or active low.

Defining a Preset/Reset Mechanism

A frequently neglected but necessary requirement is a reset mechanism. All state machine designs should have a known power up state. If a reset line is routed to all state machine registers for reset, significant routing resources will be unnecessarily used. The reset mechanism should take advantage of the hardware reset resources available in the Lattice ispLSI and pLSI device. Individual reset signals should be removed from the design equations and the hardware reset should be used. The Lattice ispLSI and pLSI devices have two reset mechanisms: a global reset for all registers and an asynchronous reset for each GLB or I/O cell.

Many high density device architectures provide only reset and no preset mechanism. Consider complementing the output requiring preset and using the hardware reset. If that is not possible, make the preset synchronous by adding a preset term into the design equations.

Circuit Partitioning

The *.DOC files produced by third party compilers are in an industry standard format. These files contain the reduced equations which are derived from the source file, JEDEC maps, high level state machine language, truth table, or standard Boolean equations. The individual PLD and SSI/MSI *.DOC files should be combined into a single source file for partitioning into the ispLSI and pLSI a device.

By grouping the equations into groups of no more than four outputs, the PLD equations can be partitioned to fit into the GLBs of the ispLSI and pLSI device since there are 4 outputs per GLB. Headers and trailers must be placed around the four equations to indicate to the Lattice pDS Software, into which GLB the equations should be loaded. The syntax is shown in table 1.

In the 1000 and 2000 family, each GLB has 18 inputs, 20 product terms and 4 registered or combinatorial outputs. Additionally, there is product term combining among the four outputs and an optional Exclusive OR gate which is fed by a single product term and an AND/OR term. The software will automatically place a given set of four equations into a GLB. The 3000 family has 24 inputs in each twin GLB (see figure 3), a programmable AND array and two OR/exclusive-OR arrays, and either outputs which can be configured to be either combinatorial or registered.

If the PLD equations do not fit into a GLB, the Lattice pDS Software will give a message as to why. If there are too many inputs, the equation can be moved into another GLB and a new equation brought into the current GLB which does not exceed the limit of 18 inputs.

As previously stated, every GLB is allowed one clock. This clock may come from either one of the four global clocks or a clock generated from a product term (.PTCLK). Ensure all registered outputs in a GLB have a single clock.

If an equation contains product terms which cannot be allocated into one GLB, consider exchanging a complex equation for one of less complexity in another GLB. If this trading of equations is not possible, simply move the equation into an empty GLB. In general, try to keep equations with common inputs in the same GLB. If a function requires a high number of product terms (product term combining), try to make use of the product term groups.

Moving a registered equation from one GLB to the next will not degrade performance as the interconnect delays between all GLBs are constant. Combinatorial equations may have an extra GLB and unit interconnect delay added to the propagation delay - if the implementation requires more than 18 inputs and 20 product terms. If an equation will not partition into a single GLB, the equation must be split into two equations and then cascaded. For Registered equations consider pipelining the intermediate equation(s) to keep the performance at the same level.

Compiling Multiple PLDs into ispLSI and pLSI Devices

The previous steps are all that are required to place PLD type designs into the GLBs of the ispLSI and pLSI devices. Note that no syntax changes of the AND/OR portions of the equations were required.

Definition of I/O Cells

The final step in the conversion process is to define the I/O cells. The basic I/O cell definition for an input and output pin is shown in listing 6 and 7 respectively. Because the device is routed according to signal names, all I/O cells will automatically be connected to the proper internal nodes. Other variations are shown in Table 2.

Table 1. Header and Trailer Syntax

Header	PLD Equations	Trailer
SYM GLB A0 <GLB NAME> 1; SIGTYPE Signal1 REG OUT; SIGTYPE Signal2 OUT; SIGTYPE Signal3 CRITICAL OUT; SIGTYPE Signal4 REG OUT; EQUATIONS	Signal1.clk=.....; Signal1=.....; Signal2=.....; Signal3=.....; Signal4=.....;	END; END;

Listing 6. Basic Input I/O Cell Definition

```
SYM IOC IOXX 1;          // IOXX = IO CELL NUMBER
XPIN IO/I X_SIG;         // IO = IO PIN; I = DEDICATED INPUT CELL
IB11 (SIG, X_SIG);       // IB = INPUT SIGNAL
END;
```

Listing 7. Basic Output I/O Cell Definition

```
SYM IOC IOXX 1;          // IOXX = IO CELL NUMBER
XPIN IO/I X_SIG;         // IO = IO PIN; I = DEDICATED INPUT CELL
OB11 (SIG, X_SIG);       // OB = OUTPUT SIGNAL
END;
```

Table 2. I/O Cell Signal Type Description

I/O Cell Type	Signal Description
IBXX	Input Pin
IDXX	Input Register
ILXX	Input D Latch
OBXX	Output Pin
OTXX	Tri-state Output Pin
BIXX	Bidirectional Pin
BIIDXX	Bidirectional Pin with Registered Input
BIILXX	Bidirectional Pin with Latched Input

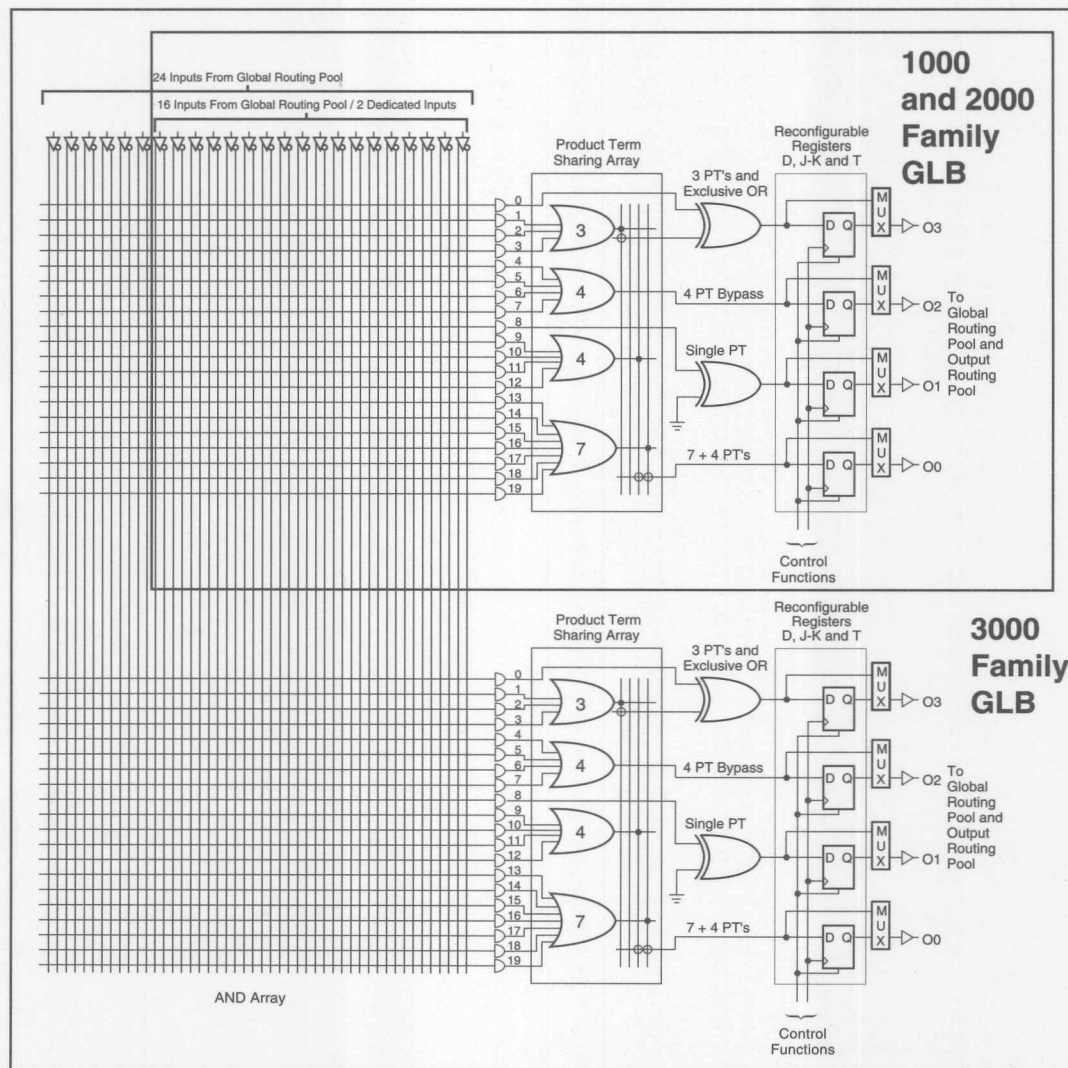
For more details refer to the pDS Development Manual under Macro Library.

Import and Verify the Design

Now that the design has been partitioned into the GLBs, the device ASCII design source file needs to be imported into the Lattice pDS Software so that it can be verified, placed, and routed. By using the FILE and IMPORT LDF commands, the ASCII file containing the design will be imported into the Lattice pDS Software. The pDS Software will check the syntax of each GLB and I/O cell and translate the ASCII file to a binary LIF (Lattice Internal Format) file.

Compiling Multiple PLDs into ispLSI and pLSI Devices

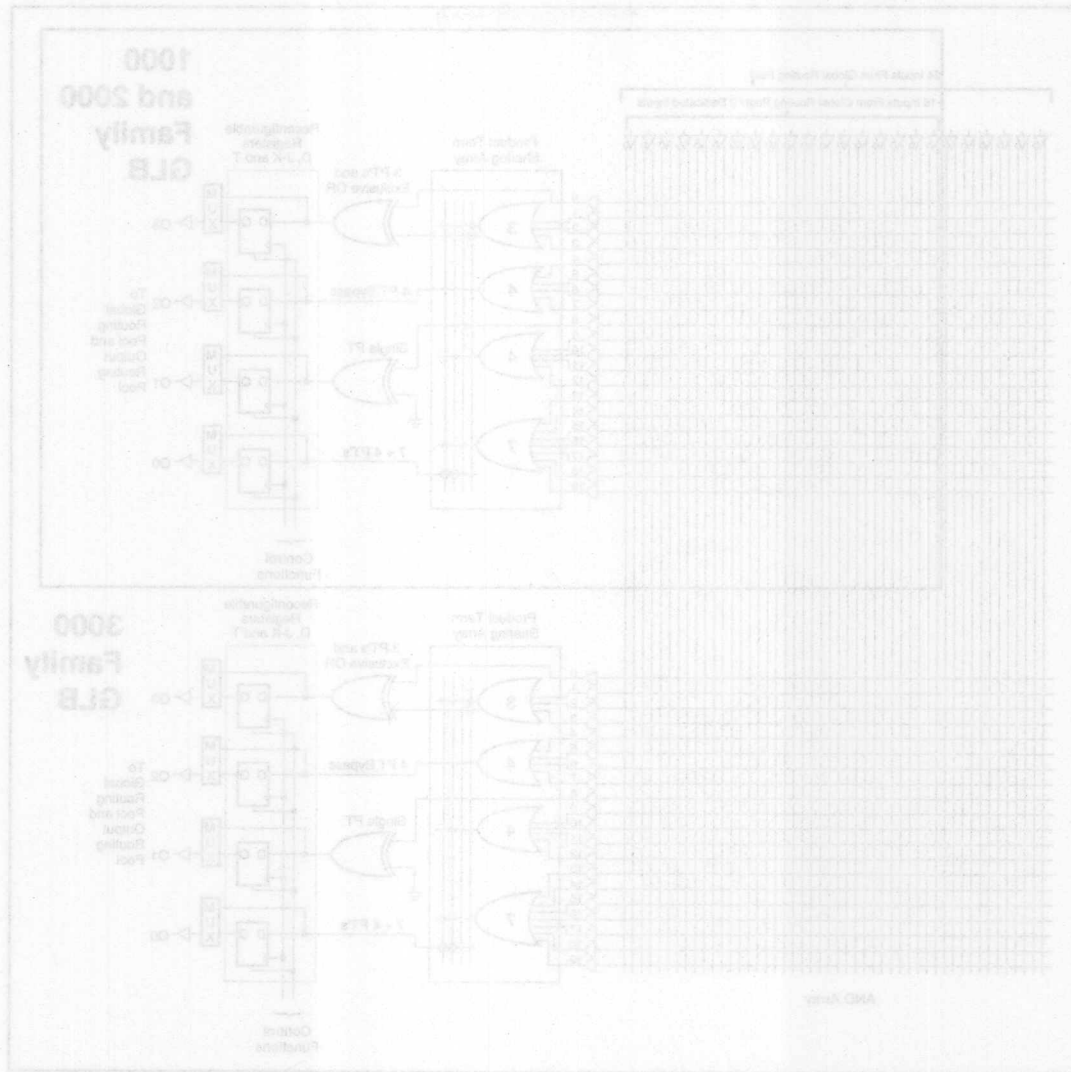
Figure 3. GLB Diagram Showing Product Term Sharing Combinations



0845

Compiling Multiple PLDs Notes

Figure 3. GLB Diagram Showing Product Term Sharing Combinations



Carry-Lookahead Adders

Arithmetic logic blocks, like adders and subtractors, are increasingly becoming performance bottlenecks in high performance logic designs. Carry-lookahead adders are generally faster than cascaded adders because they reduce the time needed to generate carry propagation. The carry-lookahead can be achieved if the input carry bit for stage i is generated directly from the inputs to the preceding stages $i-1, i-2, \dots, i-k$ rather than allowing the carry bit to be cascaded and rippled from stage to stage. An n -bit carry-lookahead adder can be constructed using k stages, each of which is a full adder stage modified by replacing its carry output line co by two signals called carry generate and propagate. These signals gi and pi are defined by the following logic:

$$gi = ai \cdot bi$$

$$pi = ai + bi$$

That is, a stage will generate a carry if both of its addend bits are 1, and it propagates carries if at least one of its addend bits is 1.

Therefore, the carry signal that will be generated for the stage $i+1$ is defined as follows from the generate and propagate signals:

$$ci + 1 = gi + pi \cdot ci$$

If we recursively expand the ci term for each stage, and multiply out to obtain a 2-level sum-of-products expression we can eliminate the carry ripple that is associated with cascaded adders. If this technique is followed, we can obtain equations for the carry out bits for each stage as shown below.

$$c1 = g0 + p0 \cdot c0$$

$$c2 = g1 + p1 \cdot c1$$

$$= g1 + p1 (g0 + p0 \cdot c0)$$

$$= g1 + p1 \cdot g0 + p1 \cdot p0 \cdot c0$$

$$c3 = g2 + p2 \cdot c2$$

$$= g2 + p2 (g1 + p1 \cdot g0 + p1 \cdot p0 \cdot c0)$$

$$= g2 + p2 \cdot g1 + p2 \cdot p1 \cdot g0 + p2 \cdot p1 \cdot p0 \cdot c0$$

$$c4 = g3 + p3 \cdot c3$$

$$= g3 + p3 (g2 + p2 \cdot g1 + p2 \cdot p1 \cdot g0 + p2 \cdot p1 \cdot p0 \cdot c0)$$

$$= g3 + p3 \cdot g2 + p3 \cdot p2 \cdot g1 + p3 \cdot p2 \cdot p1 \cdot g0 + p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0$$

Each one of the above equations corresponds to a circuit with only three levels of delay associated with it – one for the generate and propagate signals, and two for the sum-of-products shown. A carry-lookahead adder uses three-level equations such as these in each adder stage.

Building blocks of an n -bit carry-lookahead adder

F3ADD (F3ADD_1, F3ADD_2): A three bit full adder with propagate and generate outputs

PG1 .. PG4: Carry/Borrow bit generator utilizing propagate and generate inputs

F3ADD

The F3ADD macro shown below performs the 3 bit addition of $a0 \dots a2 + b0 \dots b2$, it also performs the propagate and generate functions. The propagate function determines if any of the addend bits are 1 by Oring each set of addend bits. If any of the addend bits are a one a propagate will be generated.

As shown below we see that when either of a digits addend bits are one a propagate will be generated:

	a2	a1	a0	0	1	1
	b2	b1	b0	0	0	1
propagate pi	a2+b2	a1+b1	a0+b0	0	1	1

Note: only a single propagate will be produced although more than one may be generated as shown. The generate function determines if a carry to the next digit should be generated from the previous digit addition, as shown earlier in the binary addition basics section. Basically, it performs the same function as the carry out of a regular adder, but does not incorporate the carry in signal in its logic. Shown below is an example of how a generate bit would be produced.

Adders/Subtractors in pLSI

Here we show a three bit addition with a generate being produced:

				1 1 1 0 - Carry bits from digit addition
a2	a1	a0		0 1 1 1 - 07
gi	b2	b1	b0	0 0 0 1 - 01
				1 0 0 0 - Gives 08
				Generate gi

gi will be generated because $= a0 \cdot a1 \cdot a2 \cdot b0$; here we can see that if $a0 \cdot b0 = 1$ a carry will be generated for those two addend bits. Therefore, if that carry propagates to the next set of bits $a1 \cdot b1$ and either of them are a one a carry will be generated to the next set. This function is recursive and if you look at the logic for the generate function below you will see that all combinations have been accounted for.

$$gi = (a0 \cdot a1 \cdot a2 \cdot b0$$

$$\# a1 \cdot a2 \cdot b1$$

$$\# a0 \cdot a2 \cdot b0 \cdot b1$$

$$\# a2 \cdot b2$$

$$\# a0 \cdot a1 \cdot b0 \cdot b2$$

$$\# a1 \cdot b1 \cdot b2$$

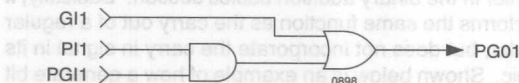
$$\# a0 \cdot b0 \cdot b1 \cdot b2)$$

PG1 .. PG4

The propagate-generate macros are carry bit generators utilizing propagate and generate inputs. Shown below is the logic and the truth table for the PG1 macro.

Figure 1. Logic and Truth Table for the PG1 Macro

PGI1	PI1	GI1	PG01
x	x	1	1
1	1	x	1



The PG1 input of the macro is the carry in input from the initial stage of your adder, the PI1, and GI1 inputs are the propagate and generate inputs associated with the F3ADD macro outputs. Looking at the logic of the PG1 macro it can be seen that whenever a generate bit was produced from the F3ADD macro a carry out signal will be generated from the PG macro, assuming the PG macro is using the inputs from the F3ADD macro outputs. This also holds true if both of the propagate inputs into the PG1 are one since the initial carry in would be a one and one of the addend bits is one, this would result in a carry out. Basically, the PG1 macro performs the function associated with the carry out of a regular adder.

As shown previously in the carry-lookahead adder section the carry signal that will be generated for the stage $i + 1$ is defined as follows from the generate and propagate signals:

$$ci + 1 = gi + pi \cdot ci \quad (\text{where } ci + 1 = PG0i, gi = GIi, pi = PII, \text{ and } ci = PGiI)$$

$$\text{Therefore: } PG0i = GIi + PII \cdot PGiI$$

If we recursively expand the PG01 term for each stage, i.e. PG02 .. PG0n and multiply out to obtain a 2-level sum-of-products expression we can eliminate the carry ripple that is associated with cascaded adders. If this technique is followed, we can obtain equations for any PG0n bits for each stage as shown previously, but substituting the generate and propagate signals into the ci equations as we did above.

For stage two (macro PG02):

$$c2 = g1 + p1 \cdot c1$$

$$= g1 + p1 (g0 + p0 \cdot c0)$$

$$= g1 + p1 \cdot g0 + p1 \cdot p0 \cdot c0$$

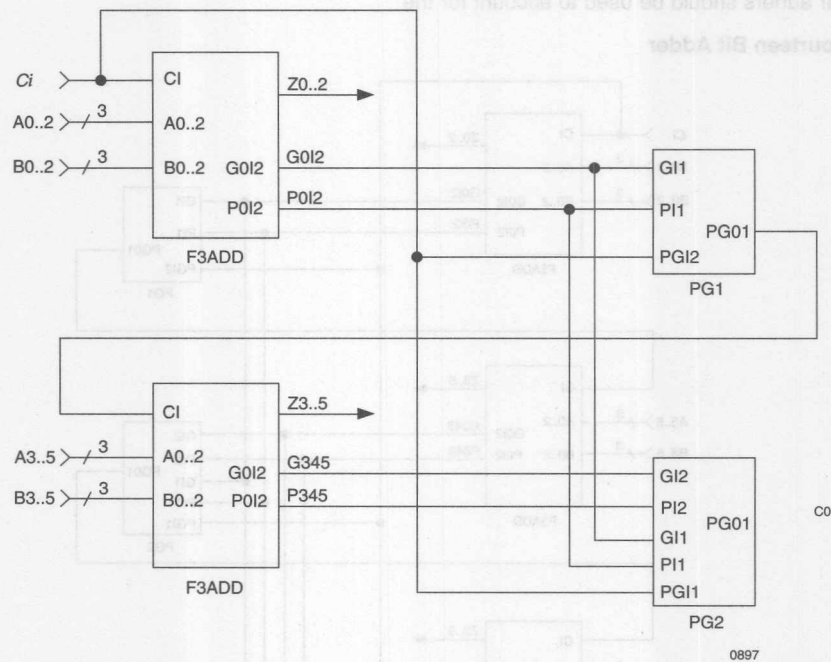
$$PG02 = GI2 + PI2 \cdot GI1 + PI2 \cdot PII \cdot PG01$$

Here we show a six bit adder utilizing two F3ADD macros, PG1 and PG2. The key thing to remember is that the propagate and generate inputs to the PGn macro is associated with that stages adders outputs.

i.e. for PG02, GI1 and PI1 would come from the first adder, and GI2 and PI2 would come from the second adder.

Adders/Subtractors in pLSI

Figure 2. Six Bit Adder

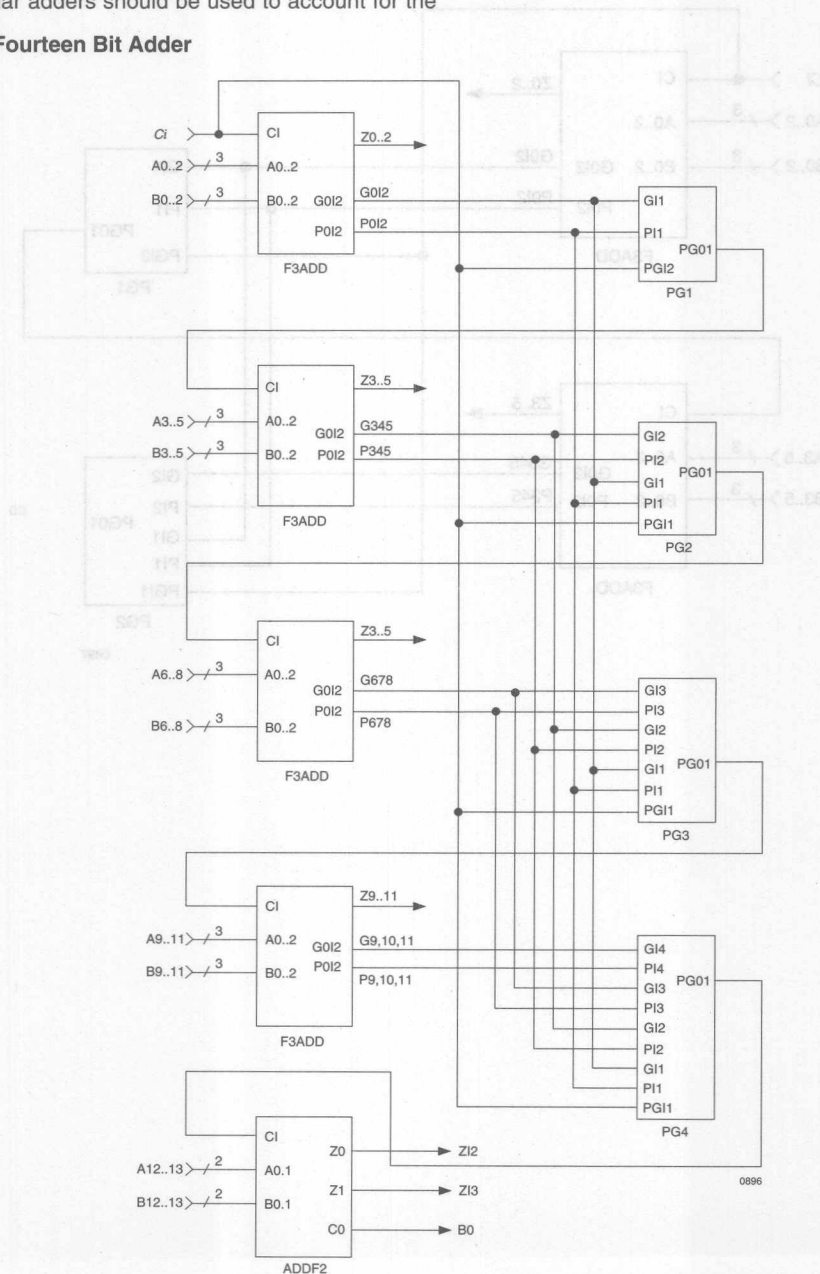


Adders/Subtractors in pLSI

In the case of the six bit adder the *co* will be the PG02 of the second propagate and generate macro PG2, but for adders that do not have a multiple of three one of the other regular adders should be used to account for the

extra bits and the *ci* of the adder would be driven by the last PG0*n* in that network. This is shown in the 14 bit adder.

Figure 3. Fourteen Bit Adder



Subtractors

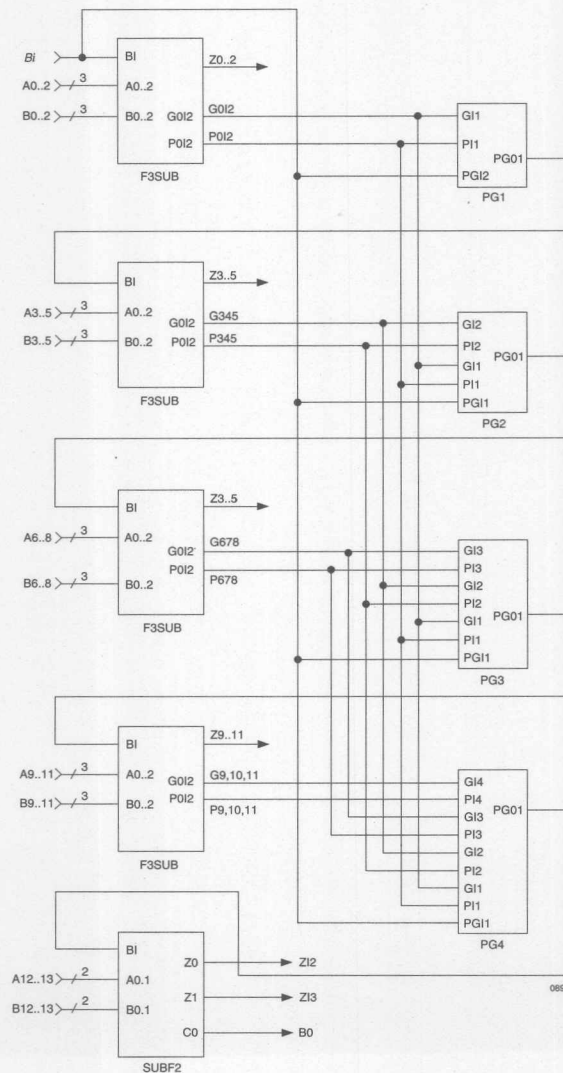
F3SUB (F3SUB_1, F3SUB_2): 3 Bit full subtractor with propagate and generate outputs

PG1 .. PG4: Carry/Borrow bit generator utilizing propagate and generate inputs

The same convention that was followed with the adders is followed with the subtractors. The only difference is

instead of having a carry in a borrow in is used. The subtraction technique is shown in the subtractor basics section, and the propagate-generate macros are identical to those used in the adder section. As shown below in the 14-bit subtractor, the borrow bit is generated by each of the PG0n macros whereas a carry bit was generated with the adders. This bit then propagates through the subtractors.

Figure 4. Fourteen Bit Subtractor

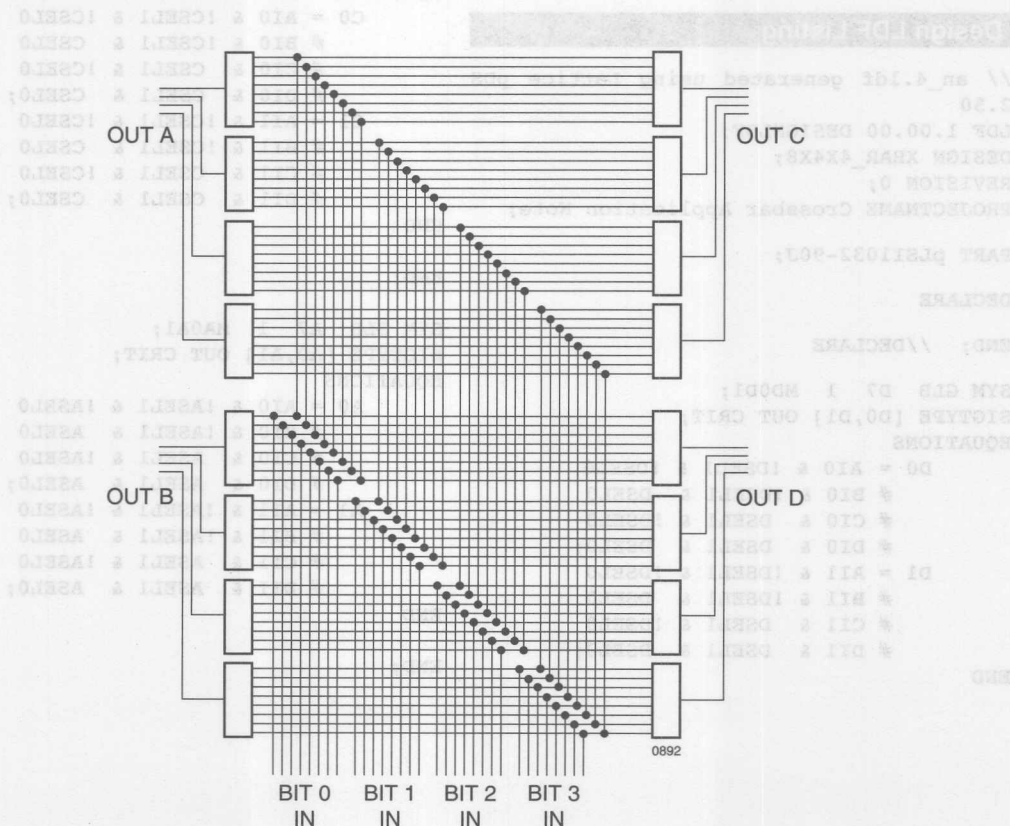


This application note describes a crosspoint switch that will allow any of four input busses to be connected to any or all of the four output busses. The input and output busses are eight bits wide. Wider busses can be accommodated by paralleling multiple pLSI 1032s. By pairing the input and output busses, the design can be changed to a two by two by sixteen crosspoint switch.

The design provides for simplex data transfers but the addition of a second device will allow duplex operation with separate transmit and receive data paths. Figure 1 shows the basic switch architecture.

The actual implementation of the switch consists of 32 4-to-1 multiplexers controlled in groups of eight. The input signal selection for each group of eight is controlled by a 2-bit register. The data written into this register is provided by an external source on the SEL0 and SEL1 pins. The address of the register to which the SEL0 and SEL1 data is to be written is provided by an external source as an address on the SELA0 and SELA1 pins. The writing of the data to the register is controlled by a write signal on the WR pin. The data stored in the register is decoded to specify the source of the data appearing at the outputs of a group of eight multiplexers.

Figure 1. Switch Architecture



Crosspoint Switch Implementation Using the pLSI 1032

A listing of the Lattice Design File (LDF) is shown on the following pages.

The signals brought in on the dedicated input pins needed to be provided to registers outside of the megablock in which the dedicated inputs were located. In order for the dedicated input signals to appear as a global signal in the global routing pool (GRP), they were routed through a generic logic block (GLB) with the output of the GLB appearing in the GRP. This implementation was used only on control paths where speed was not critical as on data paths.

The data paths all use the four product term bypass in the GLBs but do not bypass the output routing pool. The use of the four product term bypass does not affect routability in this design. The maximum propagation delay from the data input pins to the data output pins is 12ns when the Lattice pLSI 1032-90 is used.

Design LDF Listing

```
// an_4.ldf generated using Lattice pDS
2.50
LDF 1.00.00 DESIGNLDF;
DESIGN XBAR_4X4X8;
REVISION 0;
PROJECTNAME Crossbar Application Note;

PART pLSI1032-90J;

DECLARE

END; //DECLARE

SYM GLB D7 1 MD0D1;
SIGTYPE [D0,D1] OUT CRIT;
EQUATIONS
    D0 = AI0 & !DSEL1 & !DSEL0
        # BI0 & !DSEL1 & DSEL0
        # CI0 & DSEL1 & !DSEL0
        # DI0 & DSEL1 & DSEL0;
    D1 = AI1 & !DSEL1 & !DSEL0
        # BI1 & !DSEL1 & DSEL0
        # CI1 & DSEL1 & !DSEL0
        # DI1 & DSEL1 & DSEL0;

END
```

```
END;
SYM GLB B7 1 MB0B1;
SIGTYPE [B0,B1] OUT CRIT;
EQUATIONS
    B0 = AI0 & !BSEL1 & !BSEL0
        # BI0 & !BSEL1 & BSEL0
        # CI0 & BSEL1 & !BSEL0
        # DI0 & BSEL1 & BSEL0;
    B1 = AI1 & !BSEL1 & !BSEL0
        # BI1 & !BSEL1 & BSEL0
        # CI1 & BSEL1 & !BSEL0
        # DI1 & BSEL1 & BSEL0;

END
END;

SYM GLB C7 1 MC0C1;
SIGTYPE [C0,C1] OUT CRIT;
EQUATIONS
    C0 = AI0 & !CSEL1 & !CSEL0
        # BI0 & !CSEL1 & CSEL0
        # CI0 & CSEL1 & !CSEL0
        # DI0 & CSEL1 & CSEL0;
    C1 = AI1 & !CSEL1 & !CSEL0
        # BI1 & !CSEL1 & CSEL0
        # CI1 & CSEL1 & !CSEL0
        # DI1 & CSEL1 & CSEL0;

END
END;

SYM GLB A7 1 MA0A1;
SIGTYPE [A0,A1] OUT CRIT;
EQUATIONS
    A0 = AI0 & !ASEL1 & !ASEL0
        # BI0 & !ASEL1 & ASEL0
        # CI0 & ASEL1 & !ASEL0
        # DI0 & ASEL1 & ASEL0;
    A1 = AI1 & !ASEL1 & !ASEL0
        # BI1 & !ASEL1 & ASEL0
        # CI1 & ASEL1 & !ASEL0
        # DI1 & ASEL1 & ASEL0;

END
END;
```

Crosspoint Switch Implementation Using the pLSI 1032

```

SYM GLB A6 1 MA2A3;
SIGTYPE [A2,A3] OUT CRIT;
EQUATIONS
    A2 = AI2 & !ASEL1 & !ASEL0
    # BI2 & !ASEL1 & ASEL0
    # CI2 & ASEL1 & !ASEL0
    # DI2 & ASEL1 & ASEL0;
    A3 = AI3 & !ASEL1 & !ASEL0
    # BI3 & !ASEL1 & ASEL0
    # CI3 & ASEL1 & !ASEL0
    # DI3 & ASEL1 & ASEL0;
END
END;

SYM GLB B6 1 MB2B3;
SIGTYPE [B2,B3] OUT CRIT;
EQUATIONS
    B2 = AI2 & !BSEL1 & !BSEL0
    # BI2 & !BSEL1 & BSEL0
    # CI2 & BSEL1 & !BSEL0
    # DI2 & BSEL1 & BSEL0;
    B3 = AI3 & !BSEL1 & !BSEL0
    # BI3 & !BSEL1 & BSEL0
    # CI3 & BSEL1 & !BSEL0
    # DI3 & BSEL1 & BSEL0;
END
END;

SYM GLB C6 1 MC2C3;
SIGTYPE [C2,C3] OUT CRIT;
EQUATIONS
    C2 = AI2 & !CSEL1 & !CSEL0
    # BI2 & !CSEL1 & CSEL0
    # CI2 & CSEL1 & !CSEL0
    # DI2 & CSEL1 & CSEL0;
    C3 = AI3 & !CSEL1 & !CSEL0
    # BI3 & !CSEL1 & CSEL0
    # CI3 & CSEL1 & !CSEL0
    # DI3 & CSEL1 & CSEL0;
END
END;

SYM GLB D6 1 MD2D3;
SIGTYPE [D2,D3] OUT CRIT;
EQUATIONS
    D2 = AI2 & !DSEL1 & !DSEL0
    # BI2 & !DSEL1 & DSEL0
    # CI2 & DSEL1 & !DSEL0
    # DI2 & DSEL1 & DSEL0;
    D3 = AI3 & !DSEL1 & !DSEL0
    # BI3 & !DSEL1 & DSEL0
    # CI3 & DSEL1 & !DSEL0
    # DI3 & DSEL1 & DSEL0;
END
END;

SYM GLB B0 1 GSEL;
SIGTYPE [SEL0,SEL1] OUT;
EQUATIONS
    SEL0 = ISEL0;
    SEL1 = ISEL1;
END
END;

SYM GLB A5 1 MA4A5;
SIGTYPE [A4,A5] OUT CRIT;
EQUATIONS
    A4 = AI4 & !ASEL1 & !ASEL0
    # BI4 & !ASEL1 & ASEL0
    # CI4 & ASEL1 & !ASEL0
    # DI4 & ASEL1 & ASEL0;
    A5 = AI5 & !ASEL1 & !ASEL0
    # BI5 & !ASEL1 & ASEL0
    # CI5 & ASEL1 & !ASEL0
    # DI5 & ASEL1 & ASEL0;
END
END;

SYM GLB A4 1 MA6A7;
SIGTYPE [A6,A7] OUT CRIT;
EQUATIONS
    A6 = AI6 & !ASEL1 & !ASEL0
    # BI6 & !ASEL1 & ASEL0
    # CI6 & ASEL1 & !ASEL0
    # DI6 & ASEL1 & ASEL0;
    A7 = AI7 & !ASEL1 & !ASEL0
    # BI7 & !ASEL1 & ASEL0
    # CI7 & ASEL1 & !ASEL0
    # DI7 & ASEL1 & ASEL0;
END
END;

```

Crosspoint Switch Implementation Using the pLSI 1032

```

SYM GLB B5 1 MB4B5;
SIGTYPE [B4,B5] OUT CRIT;
EQUATIONS
    B4 = AI4 & !BSEL1 & !BSEL0
    # BI4 & !BSEL1 & BSEL0
    # CI4 & BSEL1 & !BSEL0
    # DI4 & BSEL1 & BSEL0;
    B5 = AI5 & !BSEL1 & !BSEL0
    # BI5 & !BSEL1 & BSEL0
    # CI5 & BSEL1 & !BSEL0
    # DI5 & BSEL1 & BSEL0;
END
END;

SYM GLB B4 1 MB6B7;
SIGTYPE [B6,B7] OUT CRIT;
EQUATIONS
    B6 = AI6 & !BSEL1 & !BSEL0
    # BI6 & !BSEL1 & BSEL0
    # CI6 & BSEL1 & !BSEL0
    # DI6 & BSEL1 & BSEL0;
    B7 = AI7 & !BSEL1 & !BSEL0
    # BI7 & !BSEL1 & BSEL0
    # CI7 & BSEL1 & !BSEL0
    # DI7 & BSEL1 & BSEL0;
END
END;

SYM GLB C5 1 MC4C5;
SIGTYPE [C4,C5] OUT CRIT;
EQUATIONS
    C4 = AI4 & !CSEL1 & !CSEL0
    # BI4 & !CSEL1 & CSEL0
    # CI4 & CSEL1 & !CSEL0
    # DI4 & CSEL1 & CSEL0;
    C5 = AI5 & !CSEL1 & !CSEL0
    # BI5 & !CSEL1 & CSEL0
    # CI5 & CSEL1 & !CSEL0
    # DI5 & CSEL1 & CSEL0;
END
END;

SYM GLB C4 1 MC6C7;
SIGTYPE [C6,C7] OUT CRIT;
EQUATIONS
    C6 = AI6 & !CSEL1 & !CSEL0
    # BI6 & !CSEL1 & CSEL0
    # CI6 & CSEL1 & !CSEL0
    # DI6 & CSEL1 & CSEL0;
    C7 = AI7 & !CSEL1 & !CSEL0
    # BI7 & !CSEL1 & CSEL0
    # CI7 & CSEL1 & !CSEL0
    # DI7 & CSEL1 & CSEL0;
END
END;

SYM GLB D5 1 MD4D5;
SIGTYPE [D4,D5] OUT CRIT;
EQUATIONS
    D4 = AI4 & !DSEL1 & !DSEL0
    # BI4 & !DSEL1 & DSEL0
    # CI4 & DSEL1 & !DSEL0
    # DI4 & DSEL1 & DSEL0;
    D5 = AI5 & !DSEL1 & !DSEL0
    # BI5 & !DSEL1 & DSEL0
    # CI5 & DSEL1 & !DSEL0
    # DI5 & DSEL1 & DSEL0;
END
END;

SYM GLB D4 1 MD6D7;
SIGTYPE [D6,D7] OUT CRIT;
EQUATIONS
    D6 = AI6 & !DSEL1 & !DSEL0
    # BI6 & !DSEL1 & DSEL0
    # CI6 & DSEL1 & !DSEL0
    # DI6 & DSEL1 & DSEL0;
    D7 = AI7 & !DSEL1 & !DSEL0
    # BI7 & !DSEL1 & DSEL0
    # CI7 & DSEL1 & !DSEL0
    # DI7 & DSEL1 & DSEL0;
END
END;

```


Crosspoint Switch Implementation Using the pLSI 1032

```

SYM GLB A3 1 CONA;
SIGTYPE [ASEL0,ASEL1] REG OUT;
EQUATIONS
    ASELO.PTCLK = !WR & !SELA0 & !SELA1;
    ASELO = SEL0;
    ASEL1 = SEL1;
END
END;

SYM GLB D0 1 GWR;
SIGTYPE WR OUT;
EQUATIONS WR = IWR;

END
END;

SYM GLB B1 1 GSELA;
SIGTYPE [SELA0,SELA1] OUT;
EQUATIONS
    SELA0 = ISELA0;
    SELA1 = ISELA1;
END
END;

SYM GLB B3 1 CONB;
SIGTYPE [BSEL0,BSEL1] REG OUT;
EQUATIONS
    BSELO.PTCLK = !WR & SELA0 & !SELA1;
    BSELO = SEL0;
    BSEL1 = SEL1;
END
END;

SYM GLB C3 1 CONC;
SIGTYPE [CSEL0,CSEL1] REG OUT;
EQUATIONS
    CSELO.PTCLK = !WR & !SELA0 & SELA1;
    CSELO = SEL0;
    CSEL1 = SEL1;
END
END;

SYM GLB D3 1 COND;
SIGTYPE [DSEL0,DSEL1] REG OUT;
EQUATIONS
    DSELO.PTCLK = !WR & SELA0 & SELA1;
    DSELO = SEL0; DSEL1 = SEL1;
END
END;

SYM IOC IO15 1 OA0;
OB11 (XA0,A);
END;

SYM IOC IO14 1 OA1;
OB11 (XA1,A1);
END;

SYM IOC IO13 1 OA2;
OB11 (XA2,A2);
END;

SYM IOC IO12 1 OA3;
OB11 (XA3,A3);
END;

SYM IOC IO31 1 OB0;
OB11 (XB0,B0);
END;

SYM IOC IO30 1 OB1;
OB11 (XB1,B1);
END;

SYM IOC IO29 1 OB2;
OB11 (XB2,B2);
END;

SYM IOC IO28 1 OB3;
OB11 (XB3,B3);
END;

SYM IOC IO47 1 OC0;
OB11 (XC0,C0);
END;

SYM IOC IO46 1 OC1;
OB11 (XC1,C1);
END;

SYM IOC IO45 1 OC2;
OB11 (XC2,C2);
END;

SYM IOC IO44 1 OC3;
OB11 (XC3,C3);
END;

SYM IOC IO63 1 OD0;
OB11 (XD0,D0);
END;

```

Crosspoint Switch Implementation Using the pLSI 1032

SYM IOC IO62 1 OD1; OB11 (XD1,D1); END;	SYM IOC IO40 1 IC3; IB11 (CI3,XCI3); END;
SYM IOC IO61 1 OD2; OB11 (XD2,D2); END;	SYM IOC IO56 1 ID3; IB11 (DI3,XDI3); END;
SYM IOC IO60 1 OD3; OB11 (XD3,D3); END;	SYM IOC IO57 1 ID2; IB11 (DI2,XDI2); END;
SYM IOC IO11 1 IA0; IB11 (AI0,XAI0); END;	SYM IOC IO58 1 ID1; IB11 (DI1,XDI1); END;
SYM IOC IO10 1 IA1; IB11 (AI1,XAI1); END;	SYM IOC IO59 1 ID0; IB11 (DI0,XDI0); END;
SYM IOC IO9 1 IA2; IB11 (AI2,XAI2); END;	SYM IOC IO0 1 IA7; IB11 (AI7,XAI7); END;
SYM IOC IO8 1 IA3; IB11 (AI3,XAI3); END;	SYM IOC IO1 1 IA6; IB11 (AI6,XAI6); END;
SYM IOC IO24 1 IB3; IB11 (BI3,XBI3); END;	SYM IOC IO2 1 IA5; IB11 (AI5,XAI5); END;
SYM IOC IO25 1 IB2; IB11 (BI2,XBI2); END;	SYM IOC IO3 1 IA4; IB11 (AI4,XAI4); END;
SYM IOC IO26 1 IB1; IB11 (BI1,XBI1); END;	SYM IOC IO4 1 OA7; OB11 (XA7,A7); END;
SYM IOC IO27 1 IB0; IB11 (BI0,XBI0); END;	SYM IOC IO5 1 OA6; OB11 (XA6,A6); END;
SYM IOC IO43 1 IC0; IB11 (CI0,XCI0); END;	SYM IOC IO6 1 OA5; OB11 (XA5,A5); END;
SYM IOC IO42 1 IC1; IB11 (CI1,XCI1); END;	SYM IOC IO7 1 OA4; OB11 (XA4,A4); END;
SYM IOC IO41 1 IC2; IB11 (CI2,XCI2); END;	SYM IOC IO16 1 IB7; IB11 (BI7,XBI7); END;

Crosspoint Switch Implementation Using the pLSI 1032

```
SYM IOC IO17 1 IB6;
IB11 (BI6,XBI6);
END;
```

```
SYM IOC IO18 1 IB5;
IB11 (BI5,XBI5);
END;
```

```
SYM IOC IO19 1 IB4;
IB11 (BI4,XBI4);
END;
```

```
SYM IOC IO20 1 OB7;
OB11 (XB7,B7);
END;
```

```
SYM IOC IO21 1 OB6;
OB11 (XB6,B6);
END;
```

```
SYM IOC IO22 1 OB5;
OB11 (XB5,B5);
END;
```

```
SYM IOC IO23 1 OB4;
OB11 (XB4,B4);
END;
```

```
SYM IOC IO32 1 IC7;
IB11 (CI7,XCI7);
END;
```

```
SYM IOC IO33 1 IC6;
IB11 (CI6,XCI6);
END;
```

```
SYM IOC IO34 1 IC5;
IB11 (CI5,XCI5);
END;
```

```
SYM IOC IO35 1 IC4;
IB11 (CI4,XCI4);
END;
```

```
SYM IOC IO36 1 OC7;
OB11 (XC7,C7);
END;
```

```
SYM IOC IO37 1 OC6;
OB11 (XC6,C6);
END;
```

```
SYM IOC IO38 1 OC5;
OB11 (XC5,C5);
END;
```

```
SYM IOC IO39 1 OC4;
OB11 (XC4,C4);
END;
```

```
SYM IOC IO48 1 ID7;
IB11 (DI7,XDI7);
END;
```

```
SYM IOC IO49 1 ID6;
IB11 (DI6,XDI6);
END;
```

```
SYM IOC IO50 1 ID5;
IB11 (DI5,XDI5);
END;
```

```
SYM IOC IO51 1 ID4;
IB11 (DI4,XDI4);
END;
```

```
SYM IOC IO52 1 OD7;
OB11 (XD7,D7);
END;
```

```
SYM IOC IO53 1 OD6;
OB11 (XD6,D6);
END;
```

```
SYM IOC IO54 1 OD5;
OB11 (XD5,D5);
END;
```

```
SYM IOC IO55 1 OD4;
OB11 (XD4,D4);
END;
```

```
SYM IOC IO 1 IS0;
IB11 (ISEL0,XSEL0);
END;
```

```
SYM IOC IO 1 IS1;
IB11 (ISEL1,XSEL1);
END;
```

```
SYM IOC IO 1 ISA0;
IB11 (ISELA0,XSELA0);
END;
```

```
SYM IOC IO 1 ISA1;
IB11 (ISELA1,XSELA1);
END;
```

```
SYM IOC IO 1 IWR;
IB11 (IWR,XWR);
END; END; //LDF DESIGNLDF
```

Notes



Building Modulo N Counters Using ispLSI and pLSI Devices

Building counters where the terminal count is not a power of two can be done using various logic configurations. Many designers simply decode the output of a binary counter and reset or load the counter when the modulo or terminal count is reached. If the reset or load is asynchronous, glitches may occur on the counter outputs. Synchronous resets and loads can eliminate the glitches but require more logic and may reduce the maximum count rate. The counter/decoder approach has an additional drawback on power up. If the counter initializes to a count higher than the decoded terminal count, the first reset or load of the counter may not occur at the proper time.

By designing the modulo n counter as a state machine with each valid state defined, the glitch problem and the long count error on initialization are eliminated. This approach allows the four product term bypass in the ispLSI and pLSI devices to be used to achieve high clock rates in prescalers and small counters.

The AND/OR/REGISTER architecture of the ispLSI and pLSI devices provides an efficient means of forcing the all ones state on the next clock edge after the terminal count is reached and forcing the counter outputs to zeroes on the clock edge following invalid output states.

Figure 1. Bit Values For a 4-Bit Up Counter

MODULO	B3	B2	B1	B0
2	0	0	0	0
3	0	0	0	1
4	0	0	1	0
5	0	0	1	1
6	0	1	0	0
7	0	1	0	1
8	0	1	1	0
9	0	1	1	1
10	1	0	0	0
11	1	0	0	1
12	1	0	1	0
13	1	0	1	1
14	1	1	0	0
15	1	1	0	1
16	1	1	1	0
No count	1	1	1	1

The table in figure 1 lists the bit values for a 4 bit up counter with the values under the "Modulo" heading indicating which states should be included for various modulo counters.

The following design example is for a modulo 11 counter. Using the table in figure 1 locate 11 under the modulo heading. The binary value to the right of the 11 indicates the terminal count that will be used to force the next state to be all ones. The counter states must include the terminal count state and all the states for lesser counts.

Figure 2. Unreduced Equations For a Modulo 11 Counter

```
B0 = !B0 & !B1 & !B2 & !B3
# !B0 & B1 & !B2 & !B3
# !B0 & !B1 & B2 & !B3
# !B0 & B1 & B2 & !B3
# !B0 & !B1 & !B2 & B3
# B0 & !B1 & !B2 & B3

B1 = B0 & !B1 & !B2 & !B3
# !B0 & B1 & !B2 & !B3
# B0 & !B1 & B2 & !B3
# !B0 & B1 & B2 & !B3
# B0 & !B1 & !B2 & B3

B2 = B0 & B1 & !B2 & !B3
# !B0 & !B1 & B2 & !B3
# B0 & !B1 & B2 & !B3
# !B0 & B1 & B2 & !B3
# B0 & !B1 & !B2 & B3

B3 = B0 & B1 & B2 & !B3
# !B0 & !B1 & !B2 & B3
# B0 & !B1 & !B2 & B3
```

The unreduced equations for a modulo 11 counter are shown in figure 2. A set of reduced equations for a modulo 11 counter are shown in figure 3. When using the pDS Software to design the counter, either the FASTMIN or STRONGMIN option should be used to reduce the product terms to four or less per output if counter speed

Building Modulo N Counters Using ispLSI and pLSI Devices

is important. If speed is not critical, additional functions can be added to the counter by adding product terms.

Figure 3. Reduced Equations For a Modulo 11 Counter

$$B0 = !B0 \& !B3$$
$$\# !B1 \& !B2 \& B3$$

$$B1 = B0 \& !B1 \& !B2$$
$$\# !B0 \& B1 \& !B3$$
$$\# B0 \& !B1 \& B2 \& !B3$$

$$B2 = B0 \& B1 \& !B2 \& !B3$$
$$\# !B1 \& B2 \& !B3$$
$$\# !B0 \& B1 \& B2 \& !B3$$
$$\# B0 \& !B1 \& !B2 \& B3$$

$$B3 = B0 \& B1 \& B2 \& !B3$$
$$\# !B1 \& !B2 \& B3$$

The reduced equations each have four product terms or less and allow the ispLSI and pLSI devices to utilize the 4 product term bypass to implement a fast counter. Counters from modulo 2 through 16 can be implemented to take advantage of the 4 product term bypass configuration. In prescaler applications, the outputs of the modulo n counter can be used to clock or enable additional counter stages to provide fast divider chains of any size. By controlling the modulo of additional stages, counters of any modulo can be constructed.

$$B2 = B0 \& B1 \& !B2 \& !B3$$
$$\# !B0 \& !B1 \& B2 \& !B3$$
$$\# B0 \& !B1 \& B2 \& !B3$$
$$\# !B0 \& B1 \& B2 \& !B3$$

$$B3 = B0 \& B1 \& B2 \& !B3$$
$$\# !B0 \& !B1 \& !B2 \& B3$$
$$\# B0 \& !B1 \& !B2 \& B3$$

The un-reduced equations for a modulo 11 counter are shown in figure 3. A set of reduced equations for a modulo 11 counter are shown in figure 3. When using the pLSI Software to design the counter, either the FASTMIN or STRONGMIN option should be used to reduce the product terms to four or less per output if counter speed

Building counters where the terminal count is not a power of two can be done using various logic configurations. Many designers simply decode the output of a binary counter and reset or load the counter when the modulo or terminal count is reached. If the reset or load is asynchronous, glitches may occur on the counter outputs. Synchronous resets and loads can eliminate the glitches but require more logic and may reduce the maximum count rate. The counter/decoder approach has an additional drawback on power up. If the counter initializes to a count higher than the decoded terminal count, the first reset or load of the counter may not occur at the proper time.

By designing the modulo n counter as a state machine with each valid state defined, the glitch problem and the long count error on initialization are eliminated. This approach allows the four product term bypass in the ispLSI and pLSI devices to be used to achieve high clock rates in prescalers and small counters.

The AND/OR REGISTER architecture of the ispLSI and pLSI devices provides an efficient means of loading the all ones state on the next clock edge after the terminal count is reached and forcing the counter outputs to zeroes on the clock edge following invalid output states.

Figure 1. Bit Values For a 4-Bit Up Counter

MODULO	B3	B2	B1	B0
2	0	0	0	0
3	0	0	0	1
4	0	0	1	0
5	0	0	1	1
6	0	1	0	0
7	0	1	0	1
8	0	1	1	0
9	0	1	1	1
10	1	0	0	0
11	1	0	0	1
12	1	0	1	0
13	1	0	1	1
14	1	1	0	0
15	1	1	0	1
16	1	1	1	0
No Count	1	1	1	1



Phase Locked Loops (PLL) in High Speed Designs

Introduction

This Application note describes the construction of a Phase Detector (PD) in conjunction with a Voltage Controlled Oscillator (VCO) to create a frequency generator synthesizer. All of the logic except the VCO and "RC" (time constant) is implemented in the ispLSI 2032 device. The logic consists of two 4-bit loadable down counters and the phase detector.

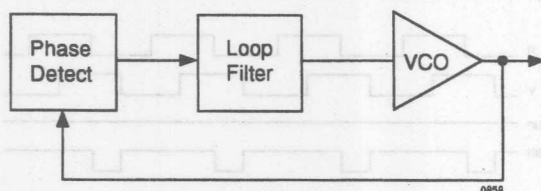
The ispLSI 2032 device has been specified because of its performance and device size. The ispLSI 2032 device is the fastest High Density Programmable Logic Device available today.

Phase Locked Loop (PLL) circuits are used in many applications ranging from communications to video and audio equipment. They are used to ensure that a clock and/or phase of that clock is stable and in sync with a reference signal.

General Information

A PLL is a circuit that consists of a phase detector, a loop filter and a reference clock. A VCO (Voltage Controlled Oscillator) is usually employed to generate the desired output frequency. Figure 1 is a block diagram of a simple PLL circuit.

Figure 1. PLL Block Diagram



When operating correctly, a PLL will "lock on" to an input and track its frequency and phase relationship. The circuit is used to synthesize or generate a frequency and maintain the phase of the generated signal to the refer-

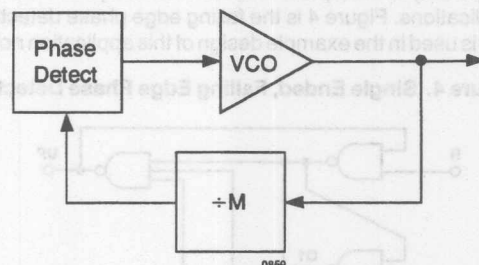
ence. It can also be used to synchronize signals (clocks) to a reference.

In the digital design world, the PLL is more accurately a phase detector. With the ability to create digital circuits that emulate analog functions, more designers are moving away from analog. Many functions can now be implemented more easily and with more flexibility due to digital design techniques.

Phase Detector

The phase detector circuit in figure 2 is analogous to an analog PLL, it could be considered a Digital Phase Locked Loop (DPLL). The results of the PLL and the DPLL will be the same, even though the method of operation between the analog and digital versions is different.

Figure 2. DPLL Block Diagram

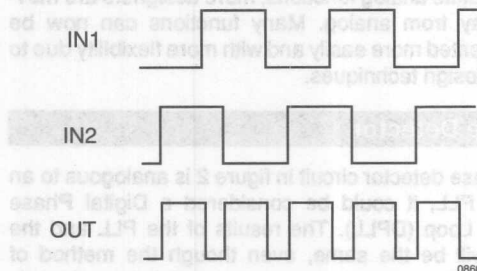


There are different types of phase detectors. A phase detector must be able to detect a change in the state of one of the two inputs and tell which input stayed constant. This is important in the basic function of the phase detector. The circuit must have the ability to detect if the reference (or the feedback signal of the PLL) changed. As a result, the phase detector will adjust its output to cause the VCO to raise or lower the frequency and phase accordingly.

Phase Lock Loops (PLL) in High Speed Designs

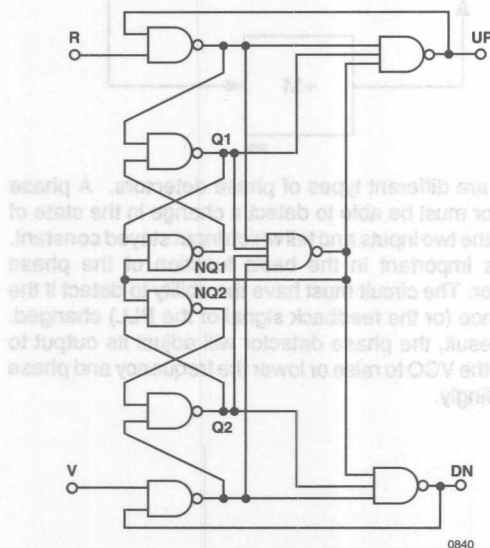
The most basic phase detector is an Exclusive Or gate (XOR). The XOR has a limited usefulness in feedback circuitry because of its inability to indicate which input changed first. Figure 3 shows the output relationship with respect to the input signals changing. This deficiency means a circuit with this type of PD would not be able to attain loop lock in some situations.

Figure 3. XOR Input/Output Wave form



A better way to implement a phase detector is with a cross-coupled latch. This single ended phase detector can be either a rising or a falling edge detector, based on the polarity of the inputs. This circuit is adequate for most applications. Figure 4 is the falling edge phase detector, and is used in the example design of this application note.

Figure 4. Single Ended, Falling Edge Phase Detector



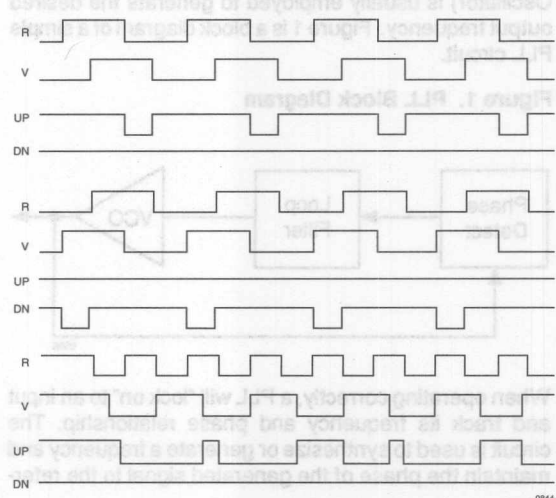
If a Rising edge version of this detector is required, the inputs can be inverted to produce the desired result.

Theory of Operation

A phase detector determines the difference in time of the edges of the two input signals. Those inputs are the reference (R) and the variable feedback (V). The difference causes the phase detector to generate pulses that cause the VCO to "correct" the frequency/phase. The loop filter is designed to allow small phase or frequency errors to be ignored. If the phase detector were to detect all changes, the PLL would go into an uncontrollable oscillation.

The PLL described in this application note uses a single ended, falling edge phase detector. This is a single ended phase detector because there is only one output for each cross-coupled NAND latch. The phase detector will detect a difference in the two input signals, however it will only react on the falling edge. The minimum phase error detected is approximately 3ns, which corresponds to the delay of the ispLSI 2032 device. The PLL will attain and remain in "loop lock" if both outputs (UP and DN) remain high. For use with a VCO, only one output is used, and the other is pulled up (if the output is an open drain). Phase error is independent of the input waveform duty cycle or its amplitude. The detector will only respond to transitions. Figure 5 shows the input and output waveform relationships of the phase detector.

Figure 5. Waveforms of the Phase Detector in figure 4

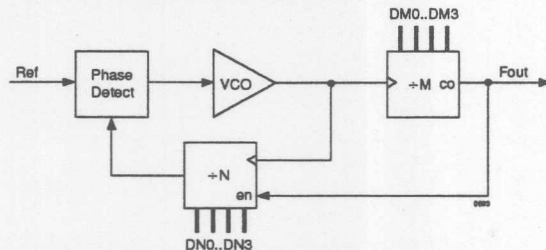


Phase Locked Loops (PLL) in High Speed Designs

Frequency Multiplier

As seen in figure 2, the DPLL has a divider which is the feedback to the phase detector. This DPLL only has the ability to generate a frequency of equal to or less than the input or reference frequency. Figure 6 is a block diagram of the frequency multiplier. By having two counters, the output of the VCO can be multiplied by a number less than, greater than, or equal to 1. This enables the output of the DPLL to be a range of frequencies less than or greater than the input. Each counter input can be brought to an external pin on the ispLSI 2032 device to preset the counters to a value (which can change) by another device such as a microprocessor. If the inputs could be eliminated, the "load value" would be fixed.

Figure 6. Frequency Multiplier



Phase Detector Equations

The following equations describe the phase detector portion of the frequency multiplier. The equations have been demorganized to show the actual implementation in the ispLSI 2032.

```
NQ2 = (!DN & Q2.PIN & Q1.PIN & !UP)
      # (Q2.PIN)
      # (!V & Q2.PIN & Q1.PIN & !R)
      # (!V & Q2.PIN & Q1.PIN & !UP)
      # (!DN & Q2.PIN & Q1.PIN & !R);

NQ1 = (Q1.PIN)
      # (!V & Q2.PIN & Q1.PIN & !R)
      # (!V & Q2.PIN & Q1.PIN & !UP)
      # (!DN & Q2.PIN & Q1.PIN & !R)
      # (!DN & Q2.PIN & Q1.PIN & !UP);

Q2 = (!NQ2.PIN)
      # (!V) # (!DN);
```

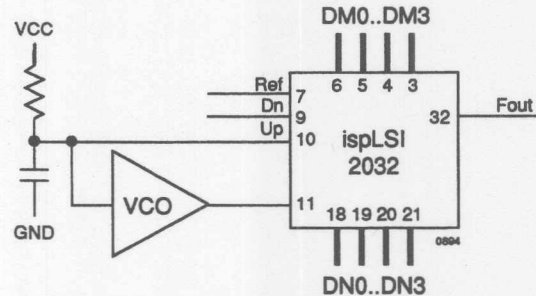
```
Q1 = (!NQ1.PIN)
      # (!R) # (!UP);

DN = (!V & Q2 & !R & Q1)
      # (!V & Q2 & !UP.PIN & Q1)
      # (V & DN.PIN)
      # (!DN.PIN & Q2 & !R & Q1)
      # (!DN.PIN & Q2 & !UP.PIN & Q1)
      # (!Q2);

UP = (!DN.PIN & Q2 & !UP.PIN & Q1)
      # (R & UP.PIN) # (!Q1)
      # (!V & Q2 & !R & Q1)
      # (!V & Q2 & !UP.PIN & Q1)
      # (!DN.PIN & Q2 & !R & Q1);
```

Figure 7 shows the pins used on the ispLSI 2032 device for the frequency multiplier. It also shows the external components needed to design the PLL.

Figure 7. ispLSI 2032 Pin Connections for the PLL Design



Summary

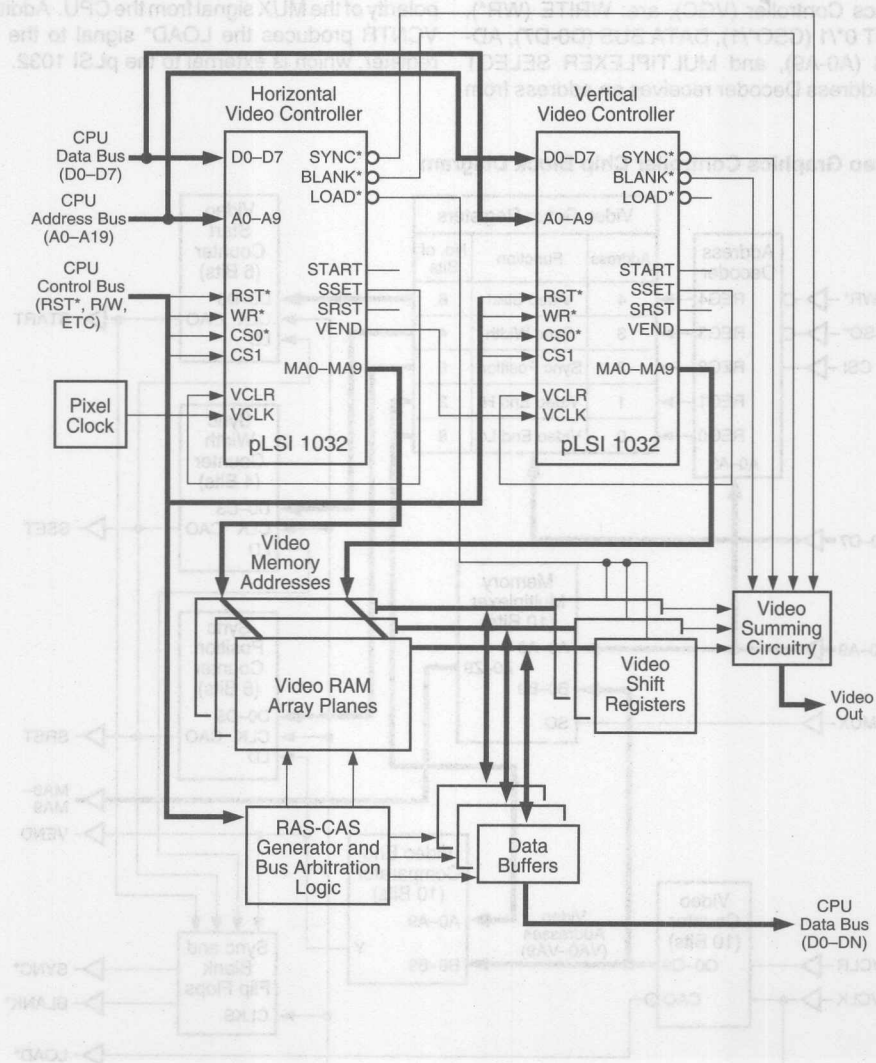
With systems and devices increasing in speed and performance, faster and more accurate clocks are required. In many situations a clock must not only be accurate, it must also have error correcting capabilities. With a DPLL users can accomplish these requirements. By using a Lattice ispLSI 2032 device, the user can also achieve these required results with greater predictability. The ispLSI 2032 also provides the user with a more accurate circuit because of its high system performance.

Introduction

This Graphics Controller design consists of two pLSI 1032 chips programmed identically to produce most of the basic video functions and timing signals associated

with a general purpose graphics interface. The generic design of the controller allows customization by adding additional circuitry for a Graphics Controller System based on the design specific requirements (see system block diagram, figure 1).

Figure 1. Video Graphics Controller System Block Diagram



Video Graphics Controller

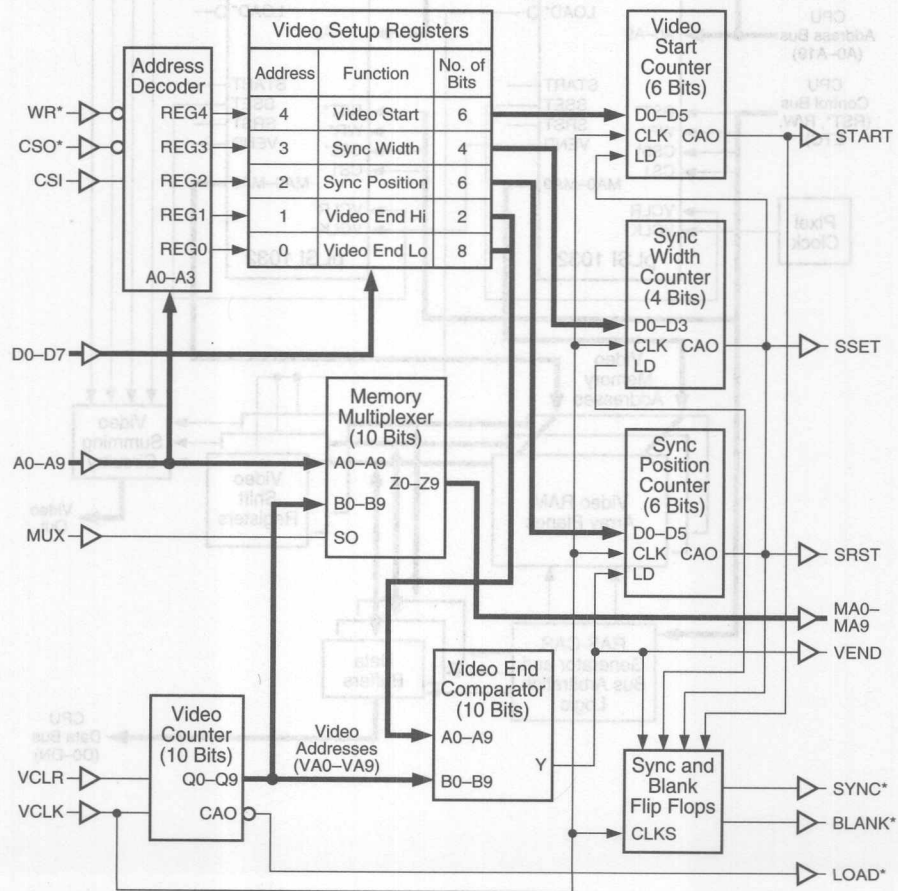
This design is capable of a maximum 1024 X 1024 non-interlaced display with programmable blanking and sync signal positioning. One of the pLSI 1032s is used for Horizontal Video Control (HVC) and the other for Vertical Video Control (VVC). Because the two pLSI 1032s are programmed identically, the LOAD* signal (Schematic 2) is redundant on the VVC chip and only used on the HVC chip.

Referencing figure 2, the Video Graphics Controller Chip block diagram, the signals which the CPU sends to the Video Graphics Controller (VGC), are: WRITE (WR*), CHIP SELECT 0*/1 (CSO*/1), DATA BUS (D0-D7), ADDRESS BUS (A0-A9), and MULTIPLEXER SELECT (MUX). The Address Decoder receives an address from

the CPU. Once decoded, this address enables one of the Video Setup Registers (VSRs) which then receives video information from the CPU data bus. This setup data is then fed to the appropriate counter or comparator, which actually controls that specific display parameter.

The CPU address bus is also interfaced to the Memory Multiplexer (MMUX) "A" inputs. The "B" inputs of the MMUX are connected to the outputs of the Video Counter (VCNTR). The MMUX allows either the CPU or the VCNTR to access video memory depending on the polarity of the MUX signal from the CPU. Additionally, the VCNTR produces the LOAD* signal to the video shift register, which is external to the pLSI 1032.

Figure 2. Video Graphics Controller Chip Block Diagram



The VCNTNTR also feeds the Video End Comparator (VEC). The VEC compares the addresses from the VCNTNTR and the Video End Hi and Lo registers which are located in the VSRs. When true, the VEC outputs the Video End (VEND) signal and simultaneously enables the load for the Sync Position Counter (SPC), while clearing the Blanking flip-flop.

The SPC data is loaded from the Sync Position register which is located in the VSRs. The SPC counts down to zero at which point it outputs the Sync Reset (SRST) signal. SRST also enables the load for the Sync Width Counter (SWC), and clears the Sync flip-flop.

The SWC's data comes from the Sync Width register in the VSRs. The SWC counts down to zero. At zero, it enables the load for the Video Start Counter (VSC), and also sets the Sync flip-flop.

The VSC receives its data from the Video Start VSR. The VSC counts down to zero, and while at zero it produces the START signal simultaneously setting the Blanking flip-flop.

1) Address Decoder (Schematic 2)

The address decoder is enabled by the WR* and CS0*/1 signals and decodes address bits A0-A2 into one of five active high select output signals, R0-R4. These are the select lines to the video attribute setup registers (schematic 3). The CS0* active low chip-select and CS1 active high chip-select are for differentiating between the horizontal controller and the vertical controller when interfacing to the CPU bus as two of these chips must be used in the system. The WR* is used to synchronize the access to the registers with CPU write cycle. All accesses to this block are write only.

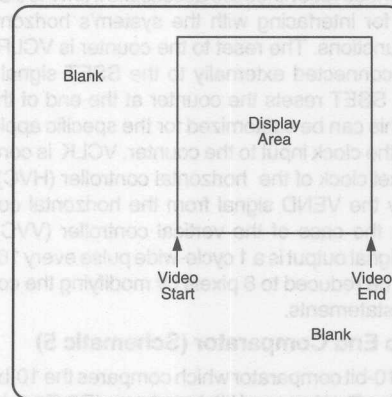
2) Video Setup Registers (Schematic 3)

The circuit is designed to interface to an 8-bit data bus but could be easily redesigned to interface to a 16-bit bus. The Video attribute Setup Register's addresses and widths are as shown in table 1.

Address	Name-Function	Number of bits
0	Video End Low (Ve 7:0)	8
1	Video End High (Ve 9:8)	2
2	Sync Position (Sp 5:0)	6
3	Sync Width (Sw 3:0)	4
4	Video Start (Vs 5:0)	6

These registers provide the data to be compared or loaded into one of the dead-end down counters used for positioning the display viewing area or sync pulse positions and widths (see figure 3).

Figure 3. Typical Video Display Set up



Video End Low and High Registers

These registers combine to form the 10-bit address location of the video display endpoints. In the case of the horizontal display location, this is the right hand side of the screen and the vertical display location is the bottom, or last visible scan line. In other words this is the point where video ends and blanking begins.

Sync Position Register

This 6-bit register holds the value of the distance from where video ends and the horizontal or vertical sync pulses start thus allowing for sync pulse positioning relative to video end. This is counted in pixels in the horizontal plane and lines in the vertical plane. The value of this register cannot be less than 1.

Sync Width Register

This 4-bit register holds the value of the sync pulse width. This is counted in pixels in the horizontal plane and lines in the vertical plane. The value of this register cannot be less than 1.

Video Start Register

This 6-bit register holds the value of the distance from where the sync pulse or blanking ends and video starts. This is counted in pixels in the horizontal plane and lines in the vertical plane. The value of this register cannot be less than 1.

Video Graphics Controller

3) Video Counter (Schematic 2)

This is a 10-bit counter which provides the video addresses VA0-VA9. In the case of the horizontal controller, this register provides the LOAD* signal for the video RAM shift registers. This register's synchronous outputs, clock, and asynchronous reset lines are accessible from the I/O pins of the chip for interfacing with the system's horizontal and vertical functions. The reset to the counter is VCLR and is typically connected externally to the SSET signal (schematic 5). SSET resets the counter at the end of the sync pulse. This can be customized for the specific application. VCLK is the clock input to the counter. VCLK is connected to the pixel clock of the horizontal controller (HVC) and is driven by the VEND signal from the horizontal controller (HVC) in the case of the vertical controller (VVC). The LOAD* signal output is a 1 cycle-wide pulse every 16 pixels. This can be reduced to 8 pixels by modifying the counter's boolean statements.

4) Video End Comparator (Schematic 5)

This is a 10-bit comparator which compares the 10-bit value in the Video End Low and High registers (R0-R1 schematic 3), to the 10-bit value of the Video Counter. When the compare is true a 1 cycle-wide pulse is generated called VEND. This is the end of visible video and starts the sync position counter running while also clearing the blanking flip-flop.

5) Sync Position Counter (Schematic 4)

This is a 6-bit loadable, dead-end down counter which counts until it reaches 0 and then holds until it is loaded with a value greater than or equal to 1. The load is activated by the VEND signal generated by the Video End Comparator. The count is a maximum of 64 pixels (horiz) or lines (vert) and is loaded each time with the value of the Sync Position Register (R2). When the count reaches zero the counter produces the signal SRST which starts the Sync Width Counter and clears the Sync flip-flop (schematic 5).

6) Sync Width Counter (Schematic 5)

This is a 4-bit loadable, dead-end down counter which counts until it reaches 0 and then holds until it is loaded with a value greater than or equal to 1. The load is activated by the SRST signal which is generated by the sync position counter. The count is a maximum of 16 pixels (horiz) or lines (vert) and is loaded each time with the value of the Sync Width Register (R3 schematic 3). When the count reaches zero the counter produces the signal SSET which starts the Video Start Counter running and sets the sync flip-flop.

7) Video Start Counter (Schematic 4)

This is a 6-bit loadable, dead-end down counter which counts until it reaches 0 and then holds until it is loaded with a value greater than or equal to 1. The load is activated by

the SSET signal (schematic 5), generated by the Sync Width Counter. The count is a maximum of 64 pixels (horiz) or lines (vert) and is loaded each time with the value of the Video Start Register (R4 schematic 3). When the count reaches zero the counter produces the signal START which sets the Blanking flip-flop (schematic 5).

8) Sync flip-flop (Schematic 5)

This flip-flop is cleared by the signal SRST (schematic 4), and set by the signal SSET to produce the sync pulse for either horizontal or vertical. It is a J-K flip-flop which is clocked by VCLK that delays the actual edges by one clock. This factor must be taken into account when calculating the sync position and sync width values as the value is one less than the true position or width. These values must be no less than 1.

9) Blanking flip-flop (Schematic 5)

This flip-flop is cleared by the signal VEND and set by the signal START (schematic 4), to produce the blanking signal for either horizontal or vertical controllers. It is a J-K flip-flop which is clocked by VCLK. This flip-flop delays the actual edges by 1 clock. This must be taken into account when calculating the sync position and sync width values as the value is one less than the true position or width. Thus the Sync position and width values must be greater than or equal to one.

10) Memory Address Multiplexer (Schematic 6)

This is a dual input 10-bit multiplexer which outputs either the video addresses (VA0-VA9), or the CPU addresses (A0-A9), to the output pins (MA0-MA9). This allows for either the video counters or the CPU to directly address the video memory. The multiplexer is controlled by the signal MUX and when MUX is low selects the CPU address. When MUX is high it selects the video counters (horizontal and vertical).

This system design is generic in terms of the size and number of the video memory planes. It is based on the additional support of RAS-CAS logic, if multiplexed dynamic RAM is used, along with bus arbitration logic to allow for transparent accesses by the CPU. It also assumes that the shift registers (if used), are correctly chosen and interfaced to the video RAM. The final support circuitry is video summing which, depending on the type of display to be driven (analog or digital), and the polarity of the blanking and sync signals has a wide variation of layouts. All of these functions, when finally chosen, can be easily incorporated into the additional 25% of each of the HVC and VVC chips remaining, or placed into additional pLSI devices as needed. This design allows for quick and flexible programmable video graphic interface to numerous applications.

Video Graphics Controller

Pin functional descriptions

NAME	TYPE	FUNCTION
WR*	Input	Allow strobe used to write data into video attribute set up register. Selected by address lines A0-A2. Also qualified with CS0*/1.
CS0*/1	Input	Active low/high chip select used to enable writes to attribute set up registers.
A0-A9	Input	A0-A2 are used to select one of the video attribute set up registers. A0-A9 are used to address the video memory.
D0-D7	Input	Data input to the video attribute set up registers.
MUX	Input	Mux select line for video memory access. High select CPU addresses (A0-A9), low select video counter addresses (VA0-VA9).
MA0-MA9	Output	Video memory address lines.
VEND	Output	Active high signal used to indicate the end of a horizontal or vertical scan.
SRST	Output	Active high signal used to indicate the end of horizontal or vertical Sync.
SSET	Output	Active high signal used to indicate the beginning of a horizontal or vertical Sync.
START	Output	Active high signal used to indicate the start of a horizontal or vertical visible scan.
LOAD*	Output	Active low signal used to load the external video shift registers with data from the video memory.
BLANK*	Output	Active low signal used to indicate the blanking of horizontal or vertical display.
SYNC*	Output	Active low signal used to indicate the horizontal or vertical Sync pulse.
VCLK	Input	System clock running at same frequency as the monitor.
VCLR	Input	Active high signal used to asynchronously reset the video counters. This allows for either horizontal or vertical operation of the device.

4

Video attribute formulas

The following are the formulas for calculating the display characteristics:

tc = pixel clock time period (ie: 10Mhz = 100ns)

Ve = video end (0-1024)

Sp = sync position (1-63)

Sw = sync width (1-15)

Vs = video start (1-63)

Horizontal (HVC)

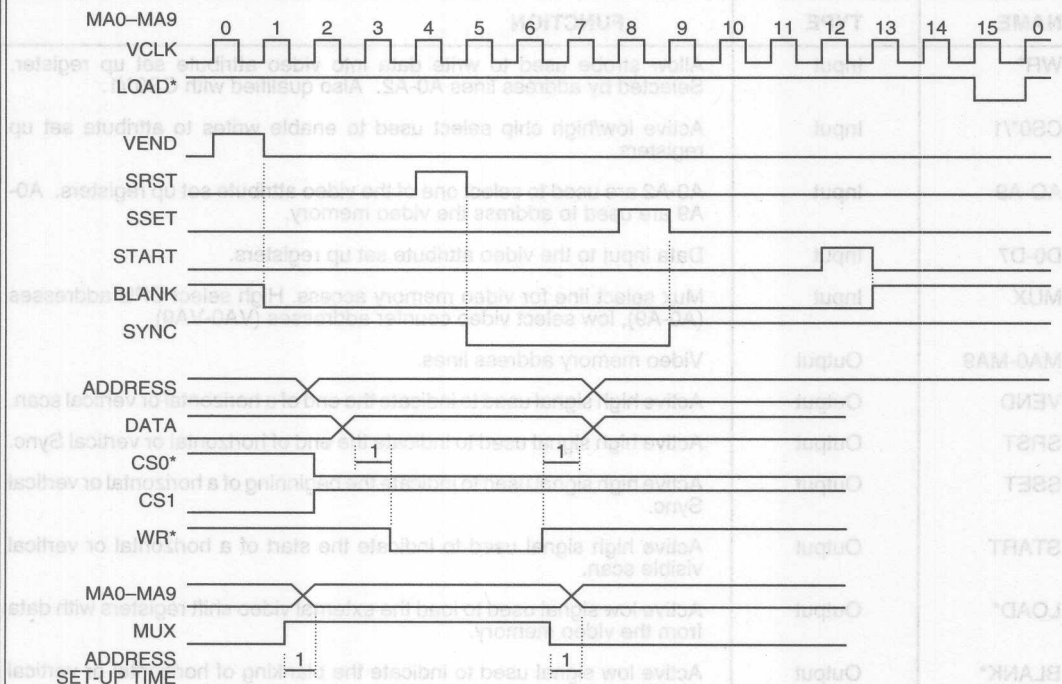
- ☐ horizontal scan line period = $[Ve + (Sp + 1) + (Sw + 1) + (Vs + 1)] * tc$
- ☐ horizontal scan rate = $1 / \text{horizontal scan line period}$
- ☐ horizontal display period = $[Ve - (Vs + 1)] * tc$
- ☐ LOAD* frequency = $tc * 1$

Vertical (VVC)

- ☐ vertical scan line period = $[Ve + (Sp + 1) + (Sw + 1) + (Vs + 1)] * \text{horizontal scan line period}$
- ☐ vertical scan rate = $1 / \text{vertical scan line period}$
- ☐ vertical display period = $[Ve - (Vs + 1)] * \text{horizontal scan line period}$

Video Graphics Controller

Figure 4. Video Graphics Controller Timing



1. Note:
See Sidebar for Description

The major timing relationships for this device are shown in figure 4. All signals are shown in relation to VCLK.

As can be seen from the diagram, LOAD* is generated every 16 VCLKs. LOAD* loads the video shift registers with data from the video memory. BLANK is activated by the falling edge of VEND and is inactivated at the falling edge of START. SYNC goes low at the falling edge of SRST and rises with the falling edge of SSET.

The CPU related signals are shown in waveforms 9 to 13. CS0* and CS1 are really complimentary versions of the same signal. Because two pLSI 1032s are used in the design, CS0*, for example, would be used as the chip select for the horizontal controller chip CS1 would then be used as the chip select for the vertical controller chip. In any case, there is a set-up and hold time associated with a data write into the chip. This is indicated by the short solid lines bounded by the dashed lines in between the DATA and CS0* waveforms. The actual set-up and hold times involved are dependent upon the frequency of VCLK, but the relationship to VCLK is clearly shown.

The last two waveforms on the diagram show the delay from MUX rising or falling and the validity of the addresses on MA0 to MA9. This delay employs the same caveat as above - the actual time depends upon the frequency of VCLK.

The pLSI Advantage

The pLSI 1032 is an excellent choice for this type of design because of its density, flexibility, and speed. The device utilization percentages for this particular design are: 75% GLB, 66% GLB output, and 61% I/O. This means that there is enough of the device left to interface to a 16-bit bus or to add glue logic which might be associated with a specific design. The I/O assignment in the pLSI 1032 is extremely flexible. I/Os can be fixed to

a specific pin, or left for the router to decide the best connection. With no fixed pins, this design took 1.5 minutes to route, and re-routing with all pins fixed was completed in a matter of seconds.

The rest of this design example consists of an appendix which contains the schematics and a hardcopy of the LDF file for this design.

Video Graphics Controller

Appendix

```
// graphfix.ldf generated using Lattice pDS Software V2.50
LDF 1.00.00 DESIGNLDF;
DESIGN GRAPHICS 1.00;
PROJECTNAME
DESCRIPTION
This is one of two identical chips used for either horizontal or vertical control
in
the graphics controller design. Two of these chips produce most of the basic
video functions and timing signals associated with a general purpose graphics
interface. The design is capable of up to a 1024 X 1024 non-interlaced display
with programmable blanking and sync signal positioning. One of the chips is
used for horizontal video control (HVC) and the other, vertical video control
(VVC).;
PART pLSI1032-90LJ;
DECLARE
END; //DECLARE

SYM GLB D4 1 MISC. SIGNALS 2;
// SSET signal generation, SYNC & BLANK;
// intermediate signal generation;
SIGTYPE SYNC REG OUT;
SIGTYPE BLANK REG OUT;
SIGTYPE SSET OUT;
EQUATIONS
    SYNC.CLK=VCLK
    SSET=!SSET1&SSET0;
    SYNC.D = !(!(!SYNC.Q & SSET) & (!SYNC.Q # SRST));
    BLANK.D = !(!(!BLANK.Q & START) & (!BLANK.Q # VEND));
END;

END;

SYM GLB C7 1 ENABLE - !WR&!CS0&CS1;
// Write enable qualification for address decoder;
SIGTYPE ENABLE OUT;
EQUATIONS
    ENABLE = !WR & !CS0 & CS1;
END;

END;

SYM GLB A1 1 VIDEO COUNTERS;
// Video memory address counter bits VA4-VA7;
SIGTYPE [VA4..VA7] REG OUT;
EQUATIONS
    VA4.CLK = VCLK;
    VA4.RE = VCLR;
    VA4=(VA0 & VA1 & VA2 & VA3) $$ VA4;
    VA5=(VA0 & VA1 & VA2 & VA3 & VA4) $$ VA5;
    VA6=(VA0 & VA1 & VA2 & VA3 & VA4 & VA5) $$ VA6;
    VA7=(VA0 & VA1 & VA2 & VA3 & VA4 & VA5 & VA6) $$ VA7;
END;

END;
```

```

SYM GLB A0 1 VIDEO COUNTERS;
// Video memory address counter bits VA0-VA3;
SIGTYPE [VA0..VA3] REG OUT;

```

EQUATIONS

```

    VA0.CLK = VCLK;
    VA0.RE = VCLR;
    VA0 = VA0 $$ VCC;
    VA1 = VA0 $$ VA1;
    VA2 = (VA0 & VA1) $$ VA2;
    VA3 = (VA0 & VA1 & VA2) $$ VA3;

```

END;

END;

```

SYM GLB A2 1 VIDEO COUNTERS;
// Video memory address counter bits VA8,VA9;
// and LOAD signal output generation;
SIGTYPE VA8 REG OUT;
SIGTYPE VA9 REG OUT;
SIGTYPE LOAD OUT;

```

EQUATIONS

```

    VA8.CLK = VCLK;
    VA8.RE = VCLR;
    VA8=(VA0 & VA1 & VA2 & VA3 & VA4 & VA5 & VA6 & VA7) $$ VA8;
    VA9=(VA0 & VA1 & VA2 & VA3 & VA4 & VA5 & VA6 & VA7 & VA8) $$ VA9;
    LOAD=VA0 & VA1 & VA2 & VA3;

```

END;

END;

```

SYM GLB A3 1 ADDRESS DECODE;
// Register address decoder;
SIGTYPE [R0..R3] OUT;

```

EQUATIONS

```

    R0 = ENABLE & !A0 & !A1 & !A2;
    R1 = ENABLE & A0 & !A1 & !A2;
    R2 = ENABLE & !A0 & A1 & !A2;
    R3 = ENABLE & A0 & A1 & !A2;

```

END;

END;

```

SYM GLB A4 1 END HI (VIDEO);
// R4 of register address decoder and video;
// data registers (video end hi);
SIGTYPE R4 OUT;
SIGTYPE [R1Q0..R1Q1] OUT;
EQUATIONS

```

```

    R4 = !WR & !CS0 & CS1 & !A0 & !A1 & A2;
    [R1Q0..R1Q1] = [D4..D5] & R1;

```

END;

END;

Video Graphics Controller

```

SYM GLB A5 1 END LO 1 (VIDEO);
// Video data registers (video end lo);
SIGTYPE [R0Q0..R0Q3] OUT;
EQUATIONS
    [R0Q0..R0Q3] = [D0..D3] & R0;
END;
END;

SYM GLB A6 1 END LO 2 (VIDEO);
// Video data registers (video end lo);
SIGTYPE [R0Q4..R0Q7] OUT;
EQUATIONS
    [R0Q4..R0Q7] = [D4..D7] & R0;
END;
END;

SYM GLB A7 1 POSITION, SYNC 1;
// Video data registers (sync position);
SIGTYPE [R2Q0..R2Q3] OUT;
EQUATIONS
    [R2Q0..R2Q3] = [D0..D3] & R2;
END;
END;

SYM GLB B0 1 START & POSITION 2;
// Video data registers (sync position);
// Video data registers (video start);
SIGTYPE [R2Q4..R2Q5] OUT;
SIGTYPE [R4Q4..R4Q5] OUT;
EQUATIONS
    [R2Q4..R2Q5] = [D4..D5] & R2;
    [R4Q4..R4Q5] = [D4..D5] & R4;
END;
END;

SYM GLB B1 1 WIDTH, SYNC;
// Video data registers (sync width);
SIGTYPE [R3Q0..R3Q3] OUT;
EQUATIONS
    [R3Q0..R3Q3] = [D0..D3] & R3;
END;
END;

SYM GLB B2 1 START, VIDEO 1;
// Video data registers (video start);
SIGTYPE [R4Q0..R4Q3] OUT;
EQUATIONS
    [R4Q0..R4Q3] = [D0..D3] & R4;
END;
END;

```



```

SYM GLB B3 1 SYNC POSITION CNTR 1;
// Low four bits of sync position counter;
SIGTYPE [Q0..Q3] REG OUT;
EQUATIONS
  [Q0..Q3].CLK = VCLK;
  Q0 = (Q0&!VEND)$$(R2Q0&VEND)#(!VEND&!SRST0));
  Q1 = (Q1&!VEND)$$(R2Q1&VEND)#(!Q0&!VEND&!SRST0));
  Q2 = (Q2&!VEND)$$(R2Q2&VEND)#(!Q0&!Q1&!VEND&!SRST0));
  Q3 = (Q3&!VEND)$$(R2Q3&VEND)#(!Q0&!Q1&!Q2&!VEND&!SRST0));
END;

SYM GLB B4 1 SYNC POSITION CNTR 2;
// Upper two bits of sync position counter;
// and sync reset signal generation;
SIGTYPE [Q4..Q5] REG OUT;
SIGTYPE SRST0 OUT;
SIGTYPE SRST1 REG OUT;
EQUATIONS
  Q4.CLK = VCLK;
  SRST1.CLK=VCLK;
  Q4 = (Q4&!VEND)$$(R2Q4&VEND)#(!Q0&!Q1&!Q2&!Q3&!VEND&!SRST0));
  Q5 = (Q5&!VEND)$$(R2Q5&VEND)#(!Q0&!Q1&!Q2&!Q3&Q4&!VEND&!SRST0));
  SRST0=!Q0&!Q1&!Q2&!Q3&!Q4&!Q5;
  SRST1.D=SRST0;
END;

SYM GLB B5 1 VIDEO START CNTR 1;
// Low four bits of video start counter;
SIGTYPE [QQ0..QQ3] REG OUT;
EQUATIONS
  [QQ0..QQ3].CLK = VCLK;
  QQ0 = (QQ0&!SSET)$$(R4Q0&SSET)#(!SSET&!START0));
  QQ1 = (QQ1&!SSET)$$(R4Q1&SSET)#(!QQ0&!SSET&!START0));
  QQ2 = (QQ2&!SSET)$$(R4Q2&SSET)#(!QQ0&!QQ1&!SSET&!START0));
  QQ3 = (QQ3&!SSET)$$(R4Q3&SSET)#(!QQ0&!QQ1&!QQ2&!SSET&!START0));
END;

SYM GLB B6 1 VIDEO START CNTR 2;
// Upper four bits of video start counter and;
// START signal generation;
SIGTYPE [QQ4..QQ5] REG OUT;
SIGTYPE START0 OUT;
SIGTYPE START1 REG OUT;
EQUATIONS
  QQ4.CLK = VCLK;
  START0=!QQ0&!QQ1&!QQ2&!QQ3&!QQ4&!QQ5;
  QQ4 = (QQ4&!SSET)$$(R4Q4&SSET)#(!QQ0&!QQ1&!QQ2&!QQ3&!SSET&!START0));

```

Video Graphics Controller

```

QQ5 =
(QQ5&!SSET)$$(R4Q5&SSET)#(!QQ0&!QQ1&!QQ2&!QQ3&QQ4&!SSET&!START0));
START1.D=START0
END;
END;

SYM GLB B7 1 SYNC WIDTH COUNTER;
// Sync width counter;
SIGTYPE [QQQ0..QQQ3] REG OUT;
EQUATIONS
[QQQ0..QQQ3].CLK = VCLK;
QQQ0 = (QQQ0&!SRST)$$(R3Q0&SRST)#(!SRST&!SSET0));
QQQ1 = (QQQ1&!SRST)$$(R3Q1&SRST)#(!QQQ0&!SRST&!SSET0));
QQQ2 = (QQQ2&!SRST)$$(R3Q2&SRST)#(!QQQ0&!QQQ1&!SRST&!SSET0));
QQQ3 = (QQQ3&!SRST)$$(R3Q3&SRST)#(!QQQ0&!QQQ1&!QQQ2&!SRST&!SSET0));
END;
END;

SYM GLB C1 1 MISC. LOGIC 1;
// Sync width counter SSet signal set-up;
// Sync reset signal generation, video START;
// signal generation;
SIGTYPE SSET0 OUT;
SIGTYPE SSET1 REG OUT;
SIGTYPE SRST OUT;
SIGTYPE START OUT;
EQUATIONS
SSET1.CLK=VCLK;
SSET0=!QQQ0&!QQQ1&!QQQ2&!QQQ3;
SSET1.D=SSET0;
SRST=!SRST1&SRST0;
START=!START1&START0;
END;
END;

SYM GLB C2 1 COMPARE, VIDEO END1;
// First eight bits of video end (VEND) comparator;
SIGTYPE VEND1 OUT;
EQUATIONS
VEND1 = !((R0Q0$VA0) # (R0Q1$VA1) # (R0Q2$VA2) # (R0Q3$VA3) # (R0Q4$VA4)
# (R0Q5$VA5) # (R0Q6$VA6) # (R0Q7$VA7));
END;
END;

SYM GLB C3 1 COMPARE, VIDEO END 2;
// Last two bits of video end (VEND) comparator;
// and VEND signal generation;
SIGTYPE VEND OUT;
EQUATIONS
VEND = !((R1Q0$VA8) # (R1Q1$VA9)) & VEND1;
END;
END;

```

```

SYM GLB C4 1 MEM ADDR MUX 1;
// Video memory address multiplexer bits;
// MA0-MA3;
SIGTYPE [MA0..MA3] OUT;
EQUATIONS
    [MA0..MA3] = ([A0..A3] & !MUX) # ([VA0..VA3] & MUX);
END;

SYM GLB C5 1 MEM ADDR MUX 2;
// Video memory address multiplexer bits;
// MA4-MA7;
SIGTYPE [MA4..MA7] OUT;
EQUATIONS
    [MA4..MA7] = ([A4..A7] & !MUX) # ([VA4..VA7] & MUX);
END;

SYM GLB C6 1 MEM ADDR MUX 3;
// Video memory address multiplexer bits;
// MA8 & MA9;
SIGTYPE [MA8,MA9] OUT;
EQUATIONS
    [MA8,MA9] = ([A8,A9] & !MUX) # ([VA8,VA9] & MUX);
END;

SYM IOC IO21 1 ;
// Read/Write control signal;
XPIN IO XWR LOCK 48 ;
IB11 (WR,XWR);
END;

SYM IOC IO20 1 ;
// Active high chip select;
XPIN IO XCS1 LOCK 3 ;
IB11 (CS1,XCS1);
END;

SYM IOC IO19 1 ;
// Active low chip select;
XPIN IO XCS0 LOCK 4 ;
IB11 (CS0,XCS0);
END;

SYM IOC IO0 1 ;
// Address bus A0;
XPIN IO XA0 LOCK 14 ;
IB11 (A0,XA0);
END;
SYM IOC IO1 1 ;

```

Video Graphics Controller

```

// Address bus A1;
XPIN IO XA1          LOCK 72 ;
IB11 (A1,XA1);
END;

SYM IOC IO2 1 ;
// Address bus A2;
XPIN IO XA2          LOCK 15 ;
IB11 (A2,XA2);
END;

SYM IOC IO3 1 ;
// Address bus A3;
XPIN IO XA3          LOCK 71 ;
IB11 (A3,XA3);
END;

SYM IOC IO4 1 ;
// Address bus A4;
XPIN IO XA4          LOCK 16 ;
IB11 (A4,XA4);
END;

SYM IOC IO5 1 ;
// Address bus A5;
XPIN IO XA5          LOCK 70 ;
IB11 (A5,XA5);
END;

SYM IOC IO6 1 ;
// Address bus A6;
XPIN IO XA6          LOCK 17 ;
IB11 (A6,XA6);
END;

SYM IOC IO7 1 ;
// Address bus A7;
XPIN IO XA7          LOCK 69 ;
IB11 (A7,XA7);
END;

SYM IOC IO8 1 ;
// Address bus A8;
XPIN IO XA8          LOCK 18 ;
IB11 (A8,XA8);
END;

SYM IOC IO9 1 ;
// Address bus A9;
XPIN IO XA9          LOCK 68 ;
IB11 (A9,XA9);
END;

SYM IOC IO10 1 ;
// Data bus D0;
XPIN IO XD0          LOCK 26 ;
IB11 (D0,XD0);
END;

SYM IOC IO11 1 ;
// Data bus D1;
XPIN IO XD1          LOCK 60 ;
IB11 (D1,XD1);
END;

SYM IOC IO12 1 ;
// Data bus D2;
XPIN IO XD2          LOCK 27 ;
IB11 (D2,XD2);
END;

SYM IOC IO13 1 ;
// Data bus D3;
XPIN IO XD3          LOCK 59 ;
IB11 (D3,XD3);
END;

SYM IOC IO14 1 ;
// Data bus D4;
XPIN IO XD4          LOCK 28 ;
IB11 (D4,XD4);
END;

SYM IOC IO15 1 ;
// Data bus D5;
XPIN IO XD5          LOCK 58 ;
IB11 (D5,XD5);
END;

SYM IOC IO16 1 ;
// Data bus D6;
XPIN IO XD6          LOCK 29 ;
IB11 (D6,XD6);
END;

SYM IOC IO17 1 ;
// Data bus D7;
XPIN IO XD7          LOCK 57 ;
IB11 (D7,XD7);
END;

SYM IOC IO18 1 ;
// Video memory address multiplexer;
XPIN IO XMUX          LOCK 55 ;
IB11 (MUX,XMUX);
END;

```

Video Graphics Controller

```
SYM IOC IO25 1 ;
// Video memory address MA0;
XPIN IO XMA0          LOCK 49 ;
OB11 (XMA0,MA0);
END;
```

```
SYM IOC IO26 1 ;
// Video memory address MA1;
XPIN IO XMA1          LOCK 79 ;
OB11 (XMA1,MA1);
END;
```

```
SYM IOC IO27 1 ;
// Video memory address MA2;
XPIN IO XMA2          LOCK 50 ;
OB11 (XMA2,MA2);
END;
```

```
SYM IOC IO28 1 ;
// Video memory address MA3;
XPIN IO XMA3          LOCK 78 ;
OB11 (XMA3,MA3);
END;
```

```
SYM IOC IO29 1 ;
// Video memory address MA4;
XPIN IO XMA4          LOCK 51 ;
OB11 (XMA4,MA4);
END;
```

```
SYM IOC IO30 1 ;
// Video memory address MA5;
XPIN IO XMA5          LOCK 77 ;
OB11 (XMA5,MA5);
END;
```

```
SYM IOC IO31 1 ;
// Video memory address MA6;
XPIN IO XMA6          LOCK 52 ;
OB11 (XMA6,MA6);
END;
```

```
SYM IOC IO32 1 ;
// Video memory address MA7;
XPIN IO XMA7          LOCK 76 ;
OB11 (XMA7,MA7);
END;
```

```
SYM IOC IO33 1 ;
// Video memory address MA8;
XPIN IO XMA8          LOCK 53 ;
OB11 (XMA8,MA8);
END;
```

```
SYM IOC IO34 1 ;
// Video memory address MA9;
XPIN IO XMA9          LOCK 75 ;
OB11 (XMA9,MA9);
END;
```

```
SYM IOC IO35 1 ;
// Video end output signal;
XPIN IO XVEND          LOCK 45 ;
OB11 (XVEND,VEND);
END;
```

```
SYM IOC IO36 1 ;
// Video sync reset signal;
XPIN IO XSRST          LOCK 46 ;
OB11 (XSRST,SRST);
END;
```

```
SYM IOC IO37 1 ;
// Video sync width set output signal;
XPIN IO XSSET          LOCK 30 ;
OB11 (XSSET,SSET);
END;
```

```
SYM IOC IO38 1 ;
// Video start output signal;
XPIN IO XSTART          LOCK 47 ;
OB11 (XSTART,START);
END;
```

```
SYM IOC IO39 1 ;
// Video load output signal;
XPIN IO XLOAD          LOCK 32 ;
OB11 (!XLOAD,!LOAD);
END;
```

```
SYM IOC IO40 1 ;
// Video Blanking output signal;
XPIN IO XBLANK          LOCK 36 ;
OB11 (XBLANK,BLANK);
END;
```

```
SYM IOC IO41 1 ;
// Video sync output signal;
XPIN IO XSYNC          LOCK 31 ;
OB11 (XSYNC,SYNC);
END;
```

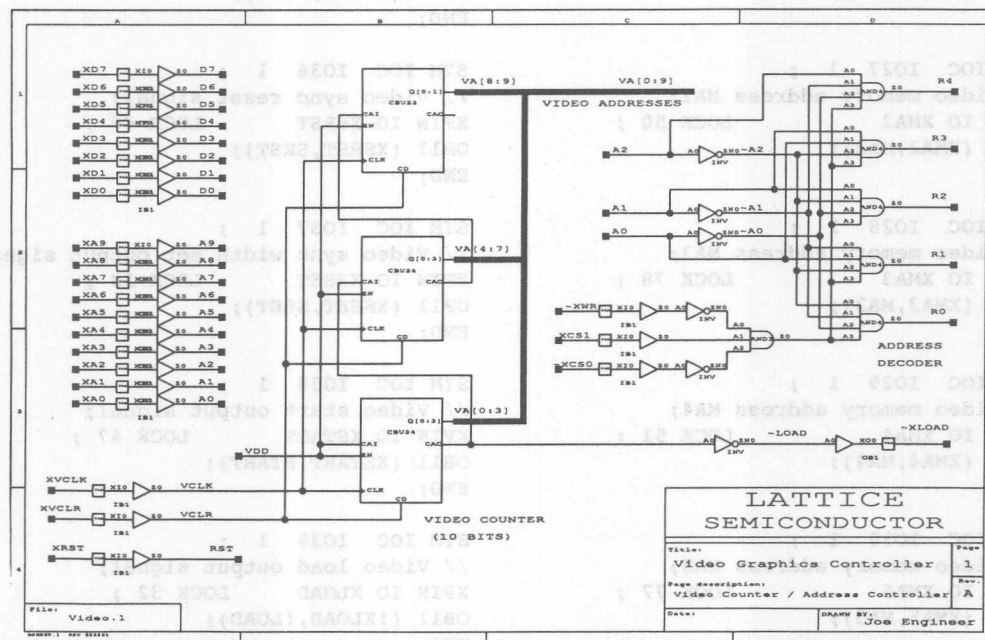
```
SYM IOC Y0 1 ;
// Video clock input signal;
XPIN CLK XVCLK          LOCK 20 ;
IB11 (VCLK,XVCLK);
END;
```


Video Graphics Controller

```

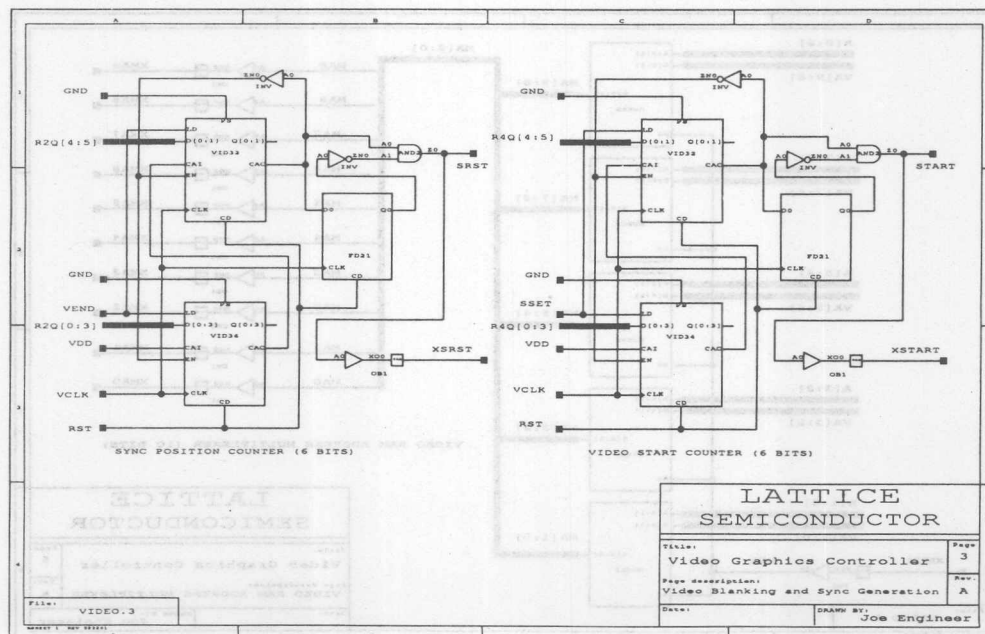
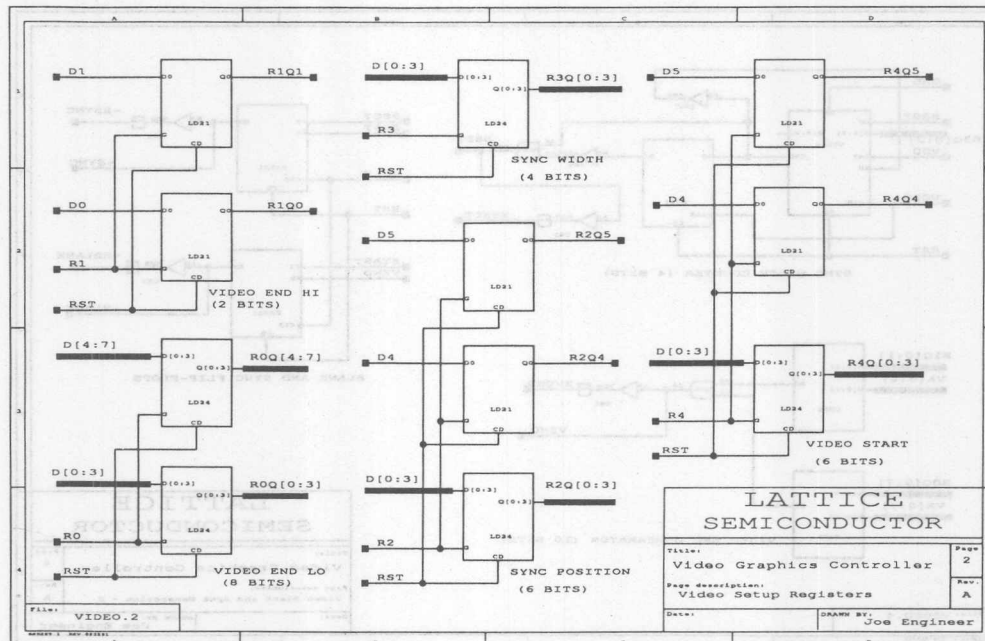
SYM IOC IO 1 ;
// Video counter clear input;
XPIN I XVCLR LOCK 25 ;
IB11 (VCLR,XVCLR);
END;
END; //LDF DESIGNLDF

```

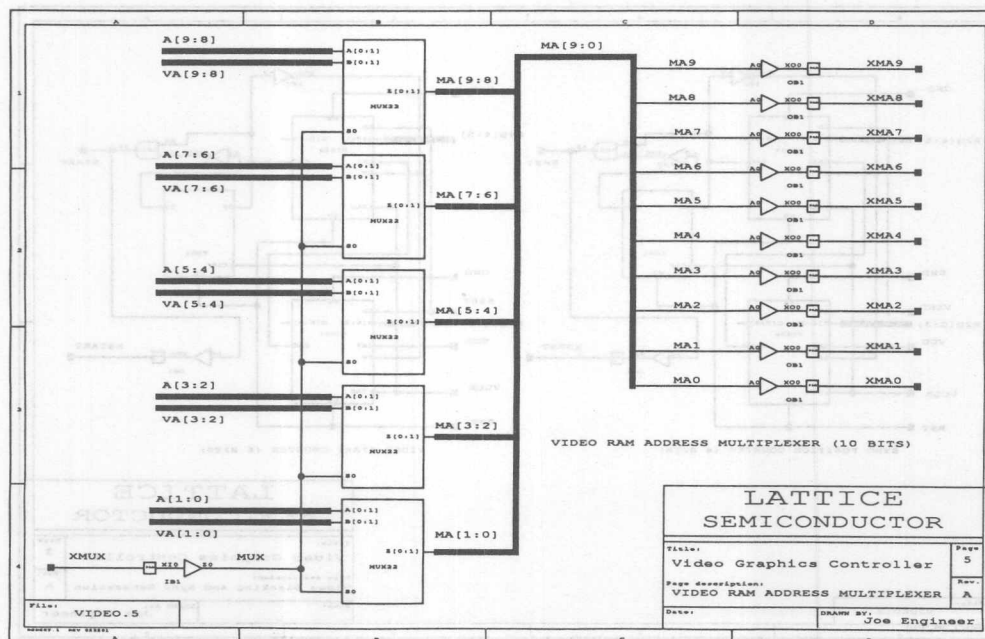
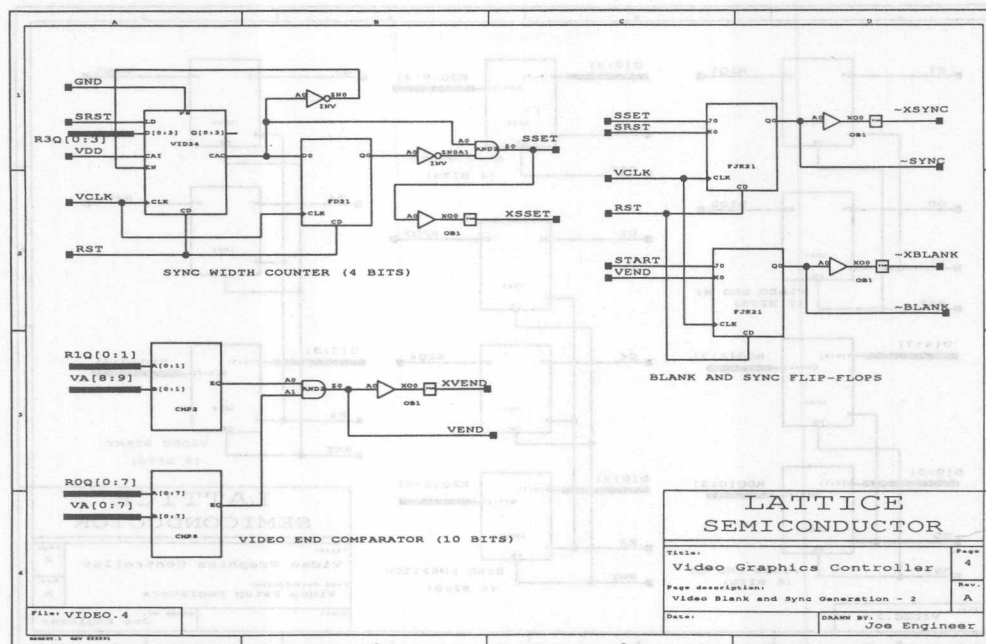


Video Graphics Controller

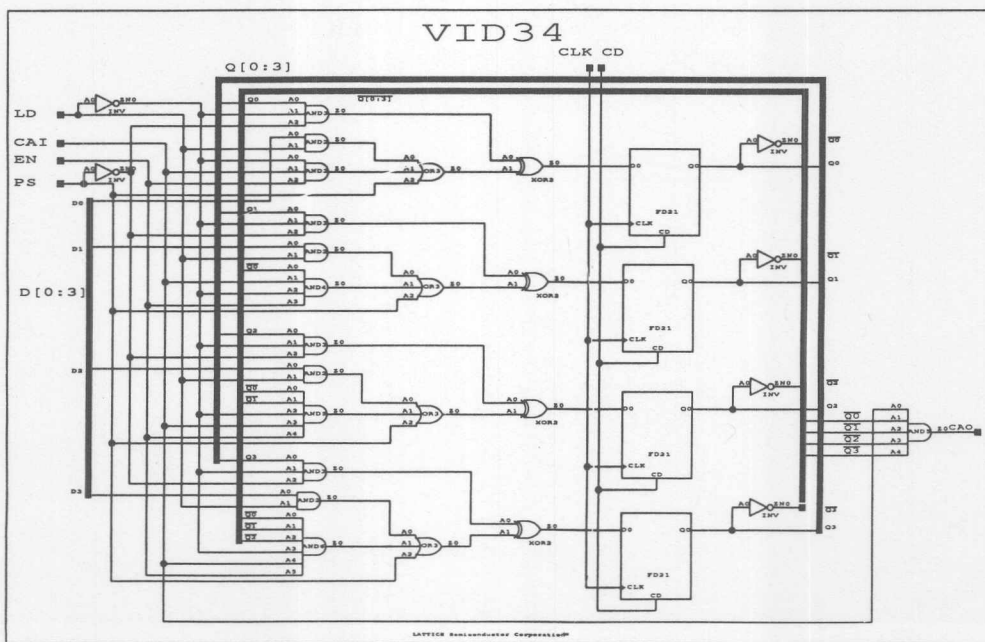
4

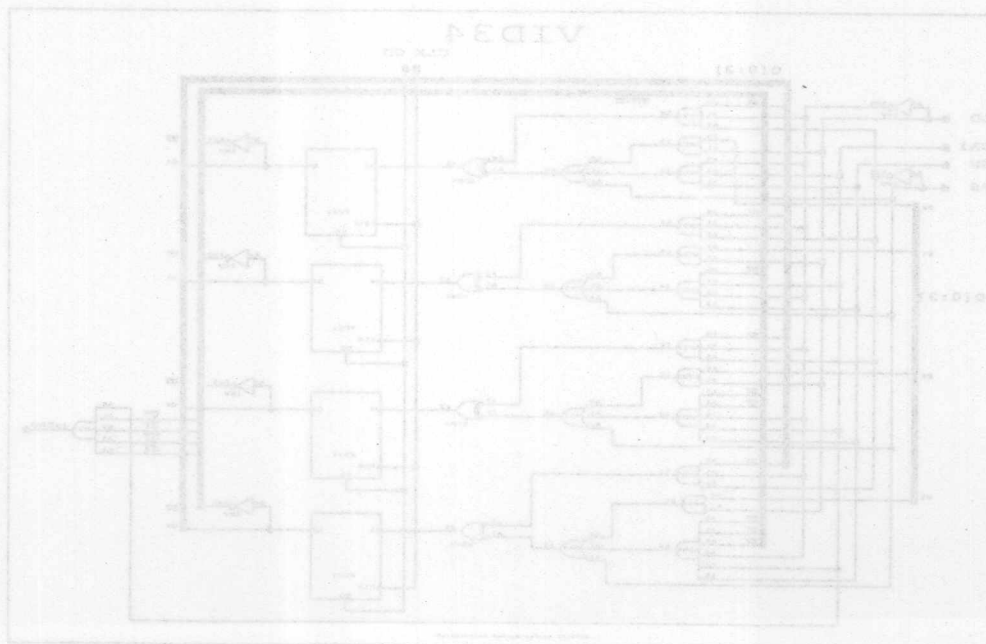
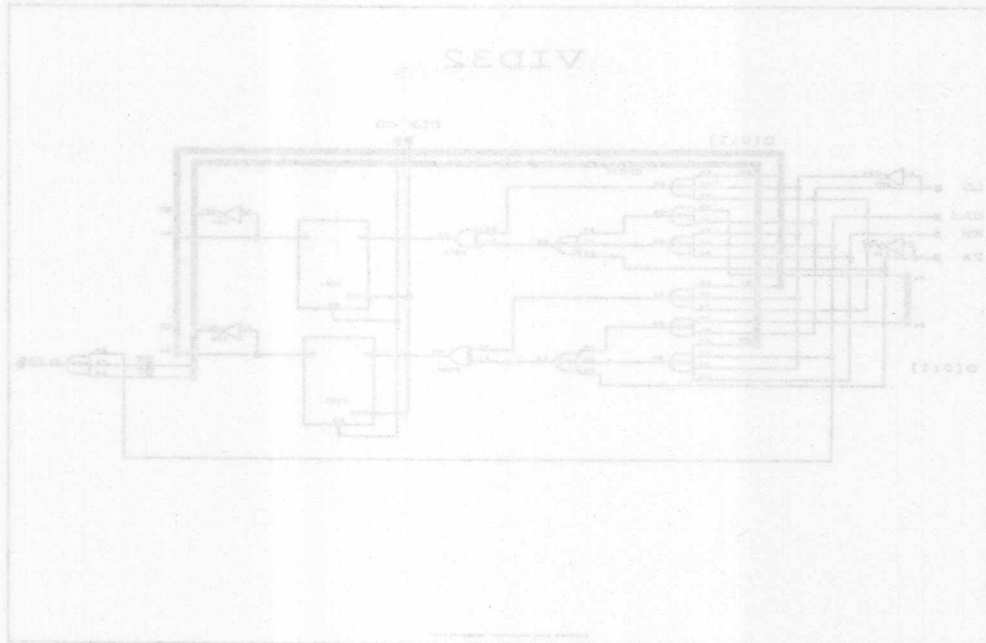


Video Graphics Controller



4

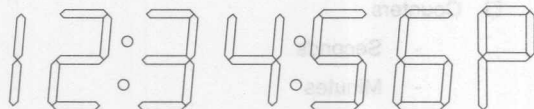




Introduction

The intent of this application note is to show how easy it is to design with an ispLSI 1032 device by implementing a simple design using many of the features of the device and design software. The digital clock was chosen because its operation is understood by most designers. This example concentrates on the design process rather than the design itself. The design also fits easily into the ispLSI 1032 demonstration box which makes it easy to debug and demonstrate. Figure 1 shows an example of a digital clock design.

Figure 1. Digital Clock Design Example



In implementing this design, advanced features are used to demonstrate the flexibility of the design environment. With the pLSI and ispLSI Development System (pDS) Software, it is simple to do a complete design using Macro library elements which are similar to parts from a typical 7400 TTL Data Book. The logic in a Macro can also be modified to meet exact design requirements. At the other extreme for complete control over the logic within the device, the circuit may be implemented with Boolean Equations. Once a custom circuit is created it can be saved as a Macro in a personal Macro library for future use.

It is assumed that the reader has read the data sheet and understands the architecture of the ispLSI device. Reading the pDS Software manuals makes it easier to understand what is being presented, but is not necessary.

The tools used in implementing this design are:

- ☐ The pDS Software Running Under Windows™ 3.1 on an IBM™ Compatible PC
- ☐ The ispDOWNLOAD Cable

Entering & Compiling the Design

Before discussing details of the clock design, the following is a quick review of the design flow. In the pDS Software, designs are created using either Boolean Equations or Macros taken from the Lattice Macro library.

Boolean logic is utilized because it is easy to use. The syntax used is similar to that used in Data I/O's ABEL™ software to design GAL® devices. With Boolean equations, designers have total control over the logic within the pLSI or ispLSI devices. Also, complete access to the advanced architectural features such as the product term Clocks and Reset, the Output Enable control, the hardware XOR is provided.

As powerful as Boolean Equations are, it is time consuming to enter a large design using them. For that reason the pDS Software comes complete with a Macro library of standard logic functions which designers can draw from. The Macro library consists of several hundred logic elements ranging from simple gates (AND, OR, XOR) to complex functions like counters, multiplexers and adders. If a standard Lattice library Macro is close to design requirements, it can be copied to a personal library and modified. This new Macro is then saved and used in other designs when needed.

For a non-standard logic function used repeatedly in a design, a Macro can be created using a combination of Boolean Equations and other Macros as described above.

Design Process

The design process in figure 2 includes the following simple steps:

- 1 Enter the Design
- 2 Verify the Logic
- 3 Route the Cells
- 4 Generate the Fusemap
- 5 Program the Part

A Digital Clock Design Example

Enter the Design

Entering the design is done using the graphical interface. The Lattice pDS Software displays a block diagram of the part similar to the one shown in the data sheet. The design equations or Macros are entered by clicking on one of the Logic or I/O Cells using the Mouse, and writing the equations into the cell using a simple text editor. This editor is similar to the Windows Notepad. The graphical interface also allows advanced functions such as clearing a cell, naming a cell, copying the contents of one cell to another or saving the data in a cell to be recalled later.

Verify the Logic

Verifying the logic is done in two places. Each GLB and I/O Cell is verified individually. A Cell Verify is a local verify of that single cell only. It checks for problems such as syntax errors, exceeding the number of allowable cell inputs, outputs or product terms, and logic errors. Once the design is completely entered, the next step is to perform a Design Verify. The Design Verify performs a Cell Verify on cells which were not previously checked, and then checks all the interconnections within the device for dangling inputs and unconnected outputs. The design must pass a Design Verify before the following steps are performed.

Route the Cells

Routing the cells is the next step. The Router module moves the GLBs and I/O Cells around in such a way that all of the networks which you have specified can be interconnected. If you have connected certain signals to specific pins, this information is entered into the design using a menu option in the Router module. Aside from fixing the I/O pins, this is an entirely automatic process and requires no intervention. Due to the optimized design of the Global Routing Pool, route times can be very fast (averaging a few minutes), depending on the size of the design and type of PC.

Generate the Fusemap

The Fusemap generation module uses the routed design to generate the JEDEC file. The JEDEC file provides the data used to program the part. This file has a suffix of .JED. Like the Router program, this is an automatic process.

Programming the Device

The part can be programmed in one of two ways. When using an external device programmer, the user can invoke a communication program to transmit the JEDEC file to the programmer. When using in-system programming (ISP) in a design, the Lattice design system invokes its own ISP control program (ISP Download). This program uses a cable connected to the PC's Parallel Port to program the part or multiple parts on the board itself.

Clock Design Description

The clock design includes the following modules:

- ☐ Control Logic
- ☐ Prescaler
- ☐ Counters
 - Seconds
 - Minutes
 - Hours

Figure 3 shows a block diagram of the clock modules.

Control Logic

The Control Logic reads the input switches and controls the speed at which the seconds, minutes, and hours are incremented. This allows a user to set the clock.

A Digital Clock Design Example

Figure 2. Design Process

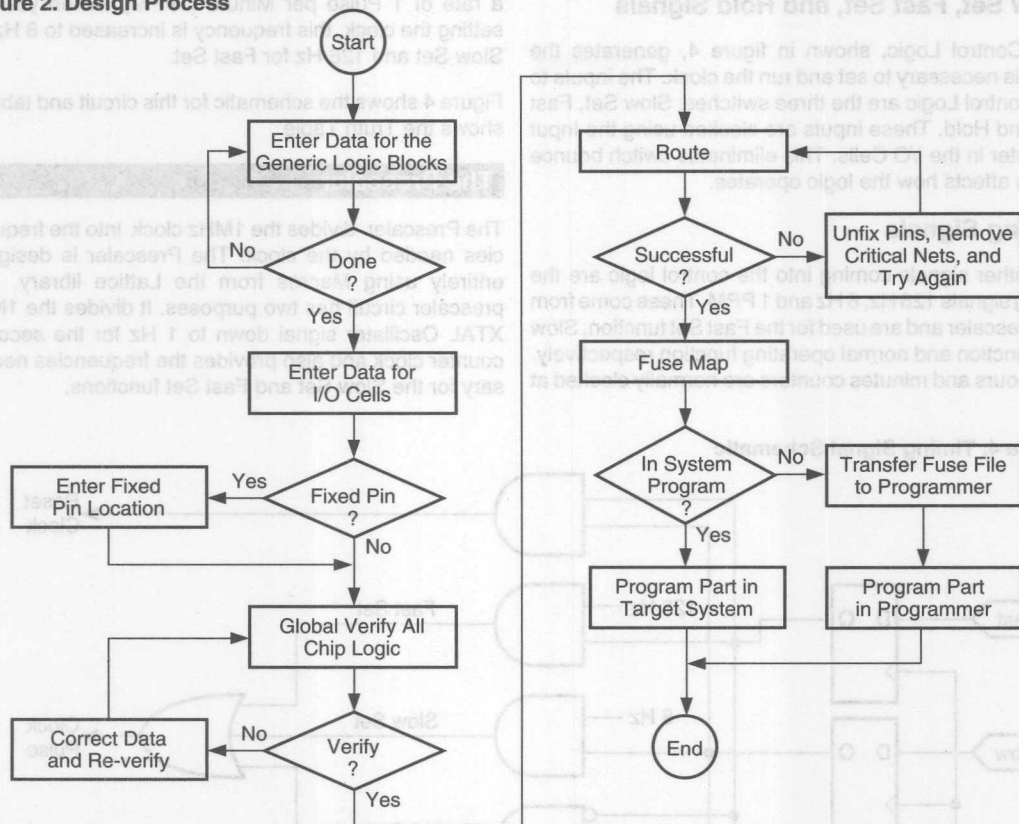
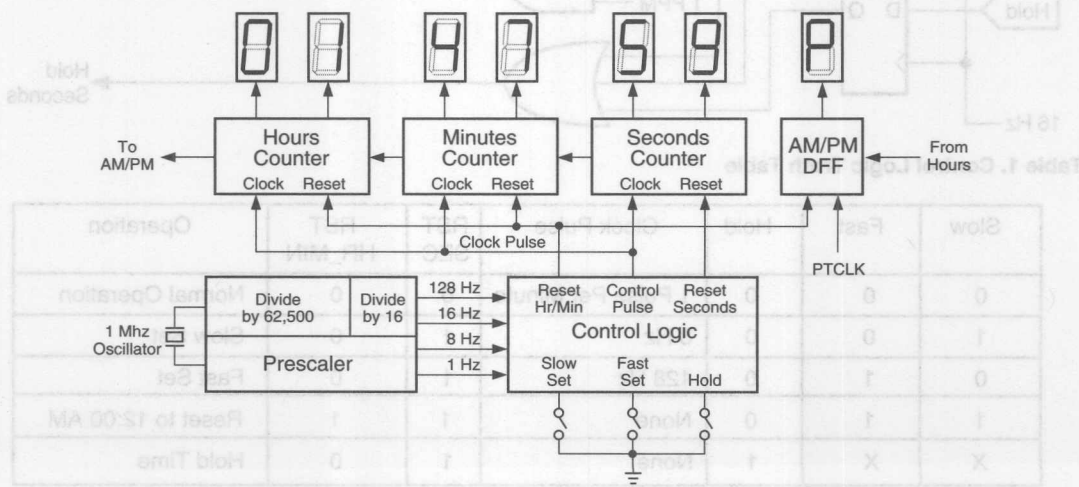


Figure 3. Block Diagram of the Clock



A Digital Clock Design Example

Slow Set, Fast Set, and Hold Signals

The Control Logic, shown in figure 4, generates the signals necessary to set and run the clock. The inputs to the Control Logic are the three switches: Slow Set, Fast Set and Hold. These inputs are clocked using the Input Register in the I/O Cells. This eliminates switch bounce which affects how the logic operates.

Timing Signals

The other signals coming into the control logic are the timing signals 128 Hz, 8 Hz and 1 PPM. These come from the prescaler and are used for the Fast Set function, Slow Set function and normal operating function respectively. The hours and minutes counters are normally clocked at

a rate of 1 Pulse per Minute (1PPM). When you are setting the clock, this frequency is increased to 8 Hz for Slow Set and 128 Hz for Fast Set.

Figure 4 shows the schematic for this circuit and table 1 shows the Truth Table.

The Prescaler

The Prescaler divides the 1MHz clock into the frequencies needed by the clock. The Prescaler is designed entirely using Macros from the Lattice library. The prescaler circuit has two purposes. It divides the 1MHz XTAL Oscillator signal down to 1 Hz for the seconds counter clock and also provides the frequencies necessary for the Slow Set and Fast Set functions.

Figure 4. Timing Signal Schematic

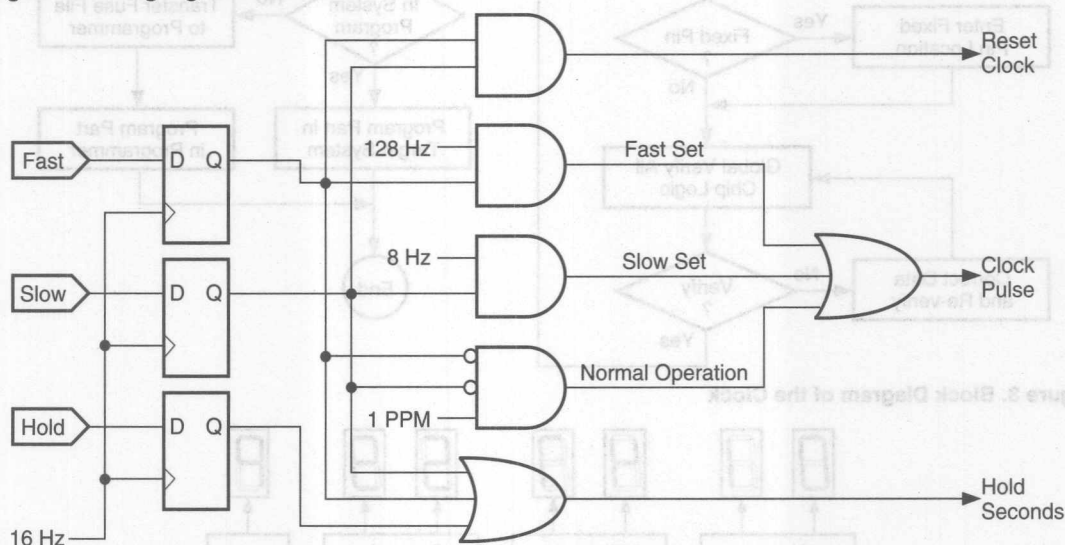


Table 1. Control Logic Truth Table

Slow	Fast	Hold	Clock Pulse	RST SEC	RST HR_MIN	Operation
0	0	0	1 Pulse Per Minute	0	0	Normal Operation
1	0	0	8 Hz	1	0	Slow Set
0	1	0	128 Hz	1	0	Fast Set
1	1	0	None	1	1	Reset to 12:00 AM
X	X	1	None	1	0	Hold Time

A Digital Clock Design Example

The circuit is implemented using two standard Macros from the Lattice library (see figure 5).

A 20-bit divider is necessary to divide the 1MHz clock signal down to 1 Hz, but the largest counter in the library is 8-bits. Therefore, three counter stages are needed to complete the division.

The approach chosen was to use two 8-bit preloadable counters and a 4-bit binary counter cascaded together. The two 8-bit counters are configured as a single 16-bit divider in this circuit. Because a binary counter was chosen for the 4-bit function, the mathematics are as follows:

$$1,000,000 \text{ Hz Divided by } 16 = 62500 \text{ Hz.}$$

Therefore, the output required of the 16-bit divider is 62500 Hz.

65535	Minus	62500	=	3035
Maximum count of the 16 bit counter	Minus	The desired Division	=	Preload Value

A 16-bit divider preloaded to 3035 (0BDB in Hexadecimal) at each terminal count has an output frequency of 16 Hz.

The frequencies necessary for the clock set functions are then chosen from the counter outputs. The 8 Hz signal (CBU14, Output Q0) advances the minutes counter at the rate of 1 minute every 7.5 Seconds. This is acceptable for the Slow Set function. The Fast Set function uses a 128Hz signal (C16Up, Output Q12) to advance the

clock at a rate of 1 Hour every 2 seconds. The 16 Hz signal is used in the I/O cell input registers as a debounce clock for the switches.

In the final design, the 16-bit counter is placed in GLBs A0 through A7, and the 4-bit counter in GLBs B0 and B1.

Counters

The Seconds and Minutes Counters are Modulo 60 counters which display a decimal count ranging from 00 to 59.

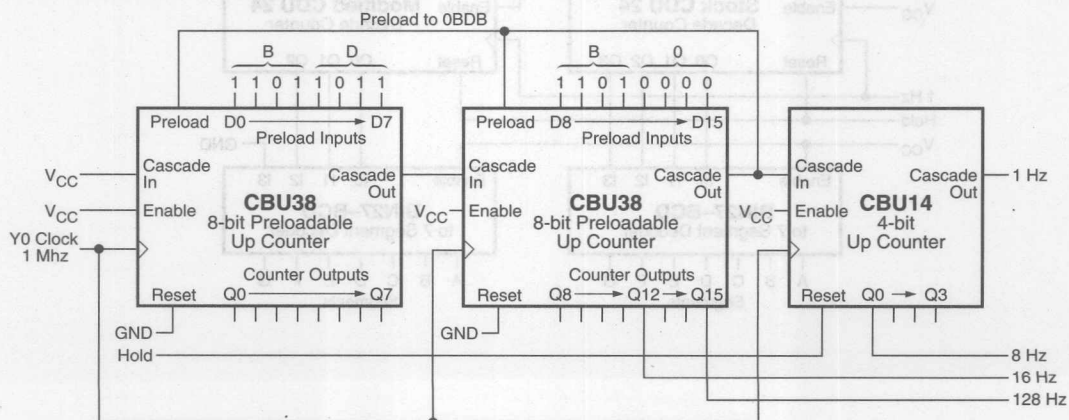
The Hours counter is a special Modulo 24 counter which counts from 1 to 12, and has a separate output bit for AM-PM indication. This counter resets to 12 AM and never displays a count of 00.

The Seconds counter is designed by using a standard Macro from the Lattice library and modifying it to suit the needs of the design. This combines the use of Macros with the use of Boolean equations.

The Minutes and Hours counters are designed using state machines optimized for the pLSI 1032 and ispLSI 1032 architectures. The counter which is created is then saved as a custom Macro for later use. This optimization saves time and effort on future designs.

There are three controls for setting the clock. These are Slow Set, Fast Set, and Hold. The Slow Set button advances the clock at a rate suitable for selecting the correct minute. The Fast Set button advances the clock at a rate suitable for selecting the correct hour. When either of these buttons are pressed, the seconds counters are reset to 00.

Figure 5. Prescaler Sample with Standard Macros



A Digital Clock Design Example

When Slow Set and Fast Set are pressed at the same time, the clock resets to 12:00 A.M.. The Hold button disables the minutes and hours counters from counting, and resets the seconds to 00 and holds that count until the button is released. This allows the clock to be set to the exact second.

The outputs from the circuit are the seven segment outputs from the Hours, Minutes and Seconds counters, and the AM/PM Indicator.

Seconds Counter

The Seconds counter is implemented using both a standard Macro from the library for the ones-of-seconds, and a modified counter Macro for the tens-of-seconds. The outputs of these counters is sent to two BCD and then to Seven Segment Display Macros to drive the LEDs.

The seconds counter counts from 0 to 59, and then resets to 0. An unmodified CDU24 decimal up counter is used for the Least Significant Digit, but the Most Significant Digit is a modulo 6 counter. This is not a standard function in the library. The easiest way to implement this function is to select a standard 4-bit binary counter (CDU24) and modify as shown in listing 3.

Listing 3.

```
MACRO MODULO6 ([Q0..Q2],CLK,EN,CS);
MACROTYPE RX;
MACROCOMMENT Custom 3 bit Modulo 6
```

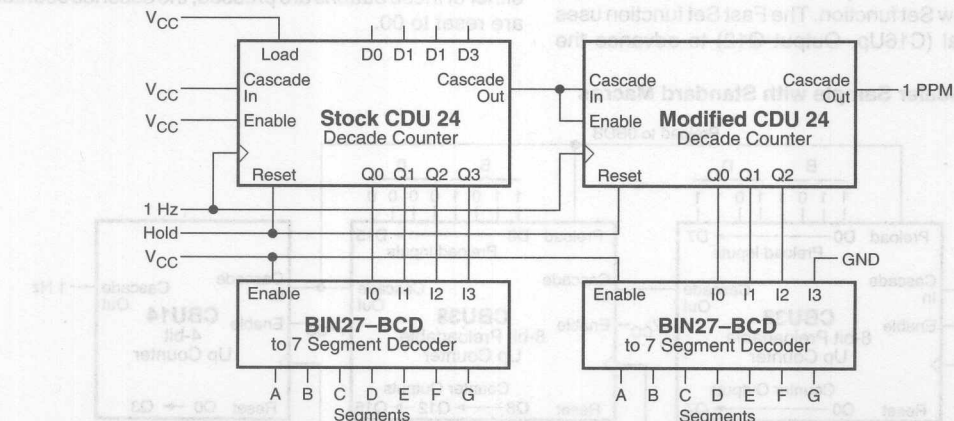
```
counter with Sync clear and enable
for clock design;
SIGTYPE [Q0..Q2] REG OUT;
EQUATIONS
Q0.CLK = CLK;
Q0 = (Q0&!EN&!CS)
    $$ (!Q0&EN&!CS);
// Output Q0 toggles after counts
// 0,2,and 4.
Q1 = (Q1&!EN&!CS)
    $$ ((!Q2&Q1&Q0&EN&!CS)
    # (!Q2&Q1&!Q0&EN&!CS));
// Output Q1 toggles after counts 1
// and 2.
Q2 = (Q2&!EN&!CS)
    $$ ((!Q2&Q1&Q0&EN&!CS)
    # (Q2&Q1&!Q0&EN&!CS));
// Output Q2 toggles after counts 3
// and 4.
END
END
```

This counter can then be saved in a personal library for future use.

The synchronous reset inputs to the seconds counters are driven by the Hold signal from the control logic. The clock is set to the exact second by setting the Hours and Minutes counters to a point several minutes ahead, and then pressing the Hold button until the correct second arrives (see figure 6).

The counters and the seven segment decoders were placed in GLBs, B2 through B7.

Figure 6. Sample Seconds Counter



A Digital Clock Design Example

A modulo 6 counter is needed for the tens-of-seconds, and it is easily created by modifying a standard Modulo 10 Counter Macro. Once that new Macro is created, it is named and saved in the personal library.

The Minutes Counter

The architecture of the Lattice ispLSI and pLSI devices has been optimized for state machine use. The registers in the GLBs are synchronous and several product terms per register are added. Each product term has 18 inputs.

In the seconds counter, since the counters and the decoders are separate, seven GLBs are used. Taking advantage of the wide input gating available to create a state machine counter which directly drives the seven segment outputs, then the number of GLBs is reduced to four. Figure 7 shows a sample minutes counter.

The truth table for a seven segment display is shown in figure 8.

The state machine is simple. The outputs are the segment drivers, and each output decodes the current state

to determine what the next state is. The simplified equation for segment A is shown in listing 4.

Listing 4. Segment A Equations

```
seg_A = seg_A & seg_B & seg_C & seg_D
        & seg_E & seg_F & !seg_G
        // Decode state Zero
# seg_A & seg_B & seg_C & seg_D
  & !seg_E & !seg_F & seg_G
  // Decode state Three
# seg_A & !seg_B & seg_C
  & seg_D & !seg_E & seg_F
  & seg_G
  // Decode state Five
```

The output for segment A goes to zero on the following clock whenever states Zero, Three or Five occur. For each of the segments there are fewer zero transitions than one. The zero transitions are decoded to save product terms, and then inverted in the output buffers. This is true on all of the segments except Segment G, which is left in its logic true form. This allows the counter to reset to a Zero when a hardware reset is applied. All segments are on except segment G.

Figure 7. Sample Minutes Counter

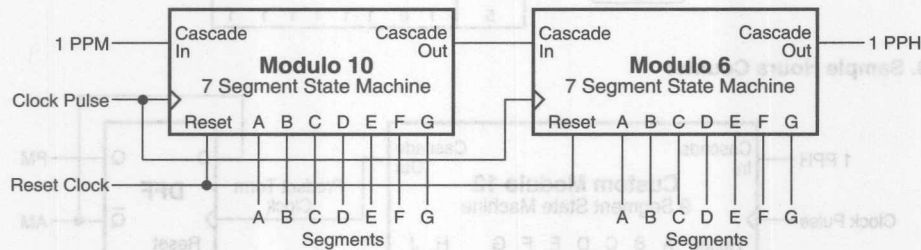


Figure 8. Seven Segment Truth Table

STATE	SEGMENT							TC
	A	B	C	D	E	F	G	
0	1	1	1	1	1	1	0	0
1	0	1	1	0	0	0	0	0
2	1	1	0	1	1	0	1	0
3	1	1	1	1	0	0	1	0
4	0	1	1	0	0	1	1	0
5	1	0	1	1	0	1	1	0
6	0	0	1	1	1	1	1	0
7	1	1	1	0	0	0	0	0
8	1	1	1	1	1	1	1	0
9	1	1	1	0	0	1	1	1

A Digital Clock Design Example

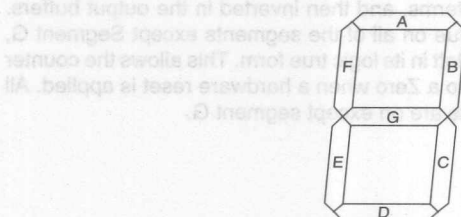
The Terminal Count (TC) output enables the next stage. The tens-of-minutes counter is similar in construction except that only the states from zero to five are decoded, and the terminal count occurs at state five instead of nine (see figure 9).

By designing the counters to make best use of the features of the pLSI device family, logic for this counter function is reduced by 40%. The minutes counters are placed in GLBs C0 through C7 in the final design.

The Hours Counter

The hours counter is constructed using a state machine similar to the one used in the minutes counter. The count sequence for hours is unique compared to most counters.

Figure 9. Terminal Count at State 5



STATE	SEGMENT							TC
	A	B	C	D	E	F	G	
0	1	1	1	1	1	0	0	0
1	0	1	1	0	0	0	0	0
2	1	1	0	1	0	1	1	0
3	1	1	1	1	0	1	1	0
4	0	1	1	0	1	1	1	0
5	1	0	1	1	1	1	1	1

Figure 10. Sample Hours Counter

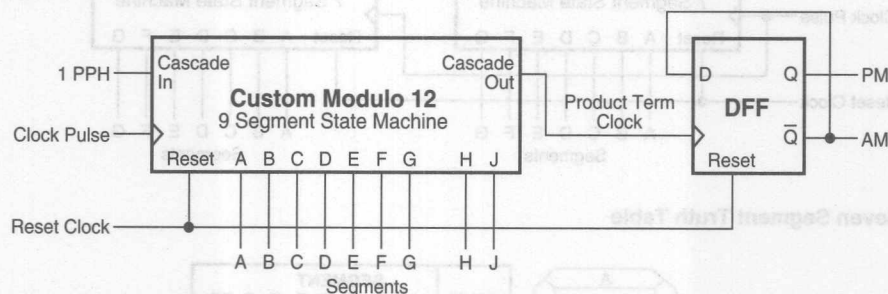


Figure 11. Sample 9 Segment Digit

STATE	SEGMENT									TC
	A	B	C	D	E	F	G	H	J	
1	0	1	1	0	0	0	0	0	0	1
2	1	1	0	1	1	0	1	0	0	1
3	1	1	1	1	0	0	1	0	0	1
4	0	1	1	0	0	1	1	0	0	1
5	1	0	1	1	0	1	1	0	0	1
6	0	0	1	1	1	1	1	0	0	1
7	1	1	1	0	0	0	0	0	0	1
8	1	1	1	1	1	1	1	0	0	1
9	1	1	1	0	0	1	1	0	0	1
10	1	1	1	1	1	0	1	1	1	1
11	0	1	1	0	0	0	0	1	1	0
12	1	1	0	1	1	0	1	1	1	1

In the hours stage, both digits are designed as a single counter stage. The reset signal for the hours stage resets the counter to 12 rather than zero (see figure 10).

A carry out signal is still generated from this counter because an AM/PM indicator is desired, but the carry out is generated when the counter reaches 12 instead of when it rolls over to one. This is consistent with the way clocks operate. Morning starts at 12:00 AM and afternoon starts at 12:00 PM. The AM/PM stage is a D-type flip-flop which uses the carry out signal as an asynchronous product term clock. This register also uses an asynchronous reset to force it to start at 12:00 AM when the clock is reset (see figure 11).

The hours counter and the AM/PM logic are placed in GLBs D0 through D4 in the example file.

A Digital Clock Design Example

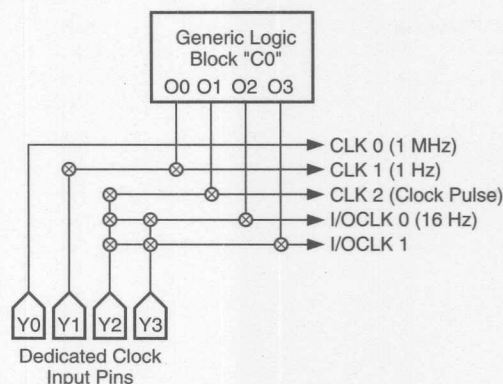
Clock Management

This design makes maximum use of the various Clock modes of the ispLSI and pLSI Family. In each GLB, there are four possible clock sources, CLK 0, CLK 1, CLK 2, and a PTCLK. The first source, CLK 0, is a synchronous clock, and is permanently connected to the Y0 Clock Input pin on the device. CLK 1 and CLK 2 are also synchronous and can come from either the external clock pins (Y1 or Y2) or can be generated within the device using the internal clock GLB, "C0". The fourth, PTCLK, comes from a Product Term within the GLB. This clock is asynchronous (see figure 12).

In this clock design, the 1MHz reference clock from the Demo Board is brought in using the Y0 Clock pin and is the clock source used to drive the Prescaler. The 1 Hz output of the Prescaler is then routed through the "C0" GLB to become the CLK 1 Source. This clock is used to increment the seconds counters. The minutes and hours counters are clocked by the signal Clock Pulse on the CLK 2 distribution line. This signal is 1 Pulse per Minute during normal operation, 8 Hz during Slow Set and 128 Hz during Fast Set Operations.

The AM/PM Indicator is a D-type flip-flop which is clocked asynchronously using a product term clock (see figure 10).

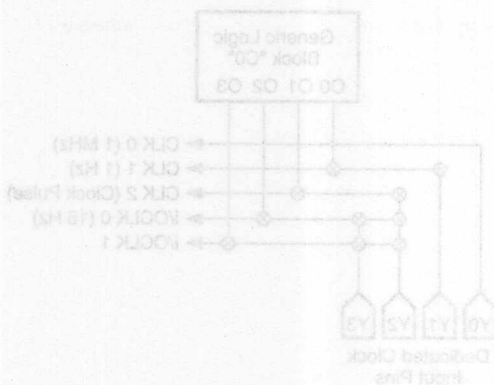
Figure 12. Clock Management Modes



Notes

The AMPM indicator is a D-type flip-flop which is clocked asynchronously using a product term clock (see figure 13).

Figure 13. Clock Management Modes



This design makes maximum use of the various Clock modes of the i96L and i96L Family. In each GCB, there are four possible clock sources: CLK 0, CLK 1, CLK 2, and a PCLK. The first source, CLK 0, is a synchronous clock, and is permanently connected to the Y0 Clock input pin on the device. CLK 1 and CLK 2 are also clock pins (Y1 or Y2) or can be generated within the device using the internal clock GCB, "00". The fourth, PCLK, comes from a Product Term within the GCB. This clock is asynchronous (see figure 13).

In this clock design, the 1MHz reference clock from the Demo Board is brought in using the Y0 Clock pin and is the clock source used to drive the Prescaler. The 1Hz output of the Prescaler is then routed through the "00" GCB to become the CLK 1 Source. This clock is used to increment the seconds counter. The minutes and hours counters are clocked by the signal Clock Pulse on the CLK 2 distribution line. This signal is 1 Pulse per Minute during normal operation, 8 Hz during Slow Set and 128 Hz during Fast Set Operations.



ispLSI Configurable Memory Controller

Introduction

There are many advantages of using the in-system programmable ispLSI devices. In board level designs, as well as during manufacturing, the flexibility of hardware reconfiguration can lead to many innovative system designs. Once configured, the ispLSI devices' non-volatile E²CMOS cells will retain their configuration even when the power is turned off. The guaranteed 1,000 programming cycles and 20 year data retention of the ispLSI device will allow the user to reliably reconfigure the device as often as required.

This application note highlights the advantages of designing with ispLSI devices and how they can lead to innovative design ideas which translate to ease of use and instant updates without board layout changes. The flexibility of design is illustrated with the use of the Dynamic Random Access Memory (DRAM) controller. This example shows a typical microprocessor and memory interface with the memory controller controlling the DRAM access and refresh timing requirements. The use of Lattice pLSI and ispLSI Development System (pDS) Software is also illustrated in this application note. The Lattice Design File (.ldf) listing file generated by the software is also attached at the end of this section.

Memory Controller Logic Overview

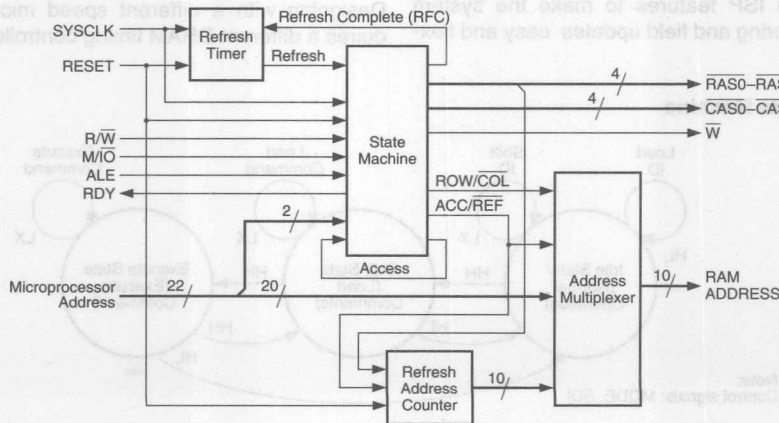
When interfacing the microprocessor to the DRAM, the control signal and timing requirements of both the proces-

sor and the DRAM must be satisfied. In order to satisfy these requirements, the external timing controller must take the processor address, data and control signals and translate them into the control signals for the DRAM. At the same time, the DRAM timing controller must take into account the refresh requirements of the DRAM.

Figure 1 shows the block diagram of the DRAM timing controller that is implemented in the ispLSI 1032. The state machine and address multiplexer blocks are used to control the memory access request of the processor and supply the DRAM with the necessary address and control signals. DRAM refresh requirements are controlled by the refresh timer block, refresh address counter block and the address multiplexer block.

Any access request from the processor is processed by the state machine based on the processor control signals such as Read/Write (R/W), Memory/IO access (M/IO), Address Latch Enable (ALE) and the microprocessor address signals. The Ready (RDY) signal is used to inform the processor the status of the requested data. In other words, it is used to acknowledge the processor that the memory is ready to respond to the processor. The address multiplexer generates the row and column addresses necessary for the memory access cycle. The appropriate Row Address Strobe (RAS), Column Address Strobe (CAS), and Write (W) signals are also generated by the state machine based on the processor

Figure 1. DRAM Timing Controller Block Diagram



ispLSI Configurable Memory Controller

inputs. To arbitrate between the memory access request and the refresh request, the state machine also generates the status signal called Access. The purpose of this signal is to keep track of an access cycle when the refresh sequence is in progress. This status signal is then used to determine whether or not to begin an access sequence after the refresh sequence. As part of the access/refresh arbitration, the state machine also issues an Access/Refresh (ACC/REF) signal to the address multiplexer logic block. Based on this signal the address multiplexer block routes the appropriate access or refresh address on to the external DRAM address bus.

As for all DRAMs, memory refresh must be completed within a specified time. This process is completely controlled by the DRAM timing controller. The refresh timer block generates the internal refresh request signal according to the system clock speed and the DRAM refresh rate requirements. When the state machine detects this refresh request signal, the refresh sequence for the DRAM is generated as soon as time permits. This means that the refresh sequence is generated right after the refresh request or if the timing controller is in the middle of a memory access cycle the refresh sequence is generated right after the memory access cycle is complete. During the refresh sequence the row address and all the RAS signal must be activated to perform the basic RAS-only refresh. The row addresses are supplied by the refresh address counter logic block. This logic block keeps track of the rows that are being refreshed and it gets incremented every time a refresh sequence is performed. All the RAS signal are activated for refresh by the state machine.

With the basic understanding of the DRAM timing control logic complete, the next section will discuss the implementation of the logic in an ispLSI device and how to take advantage of the ISP features to make the system design, manufacturing and field updates easy and flexible.

Taking Advantage of ISP Features

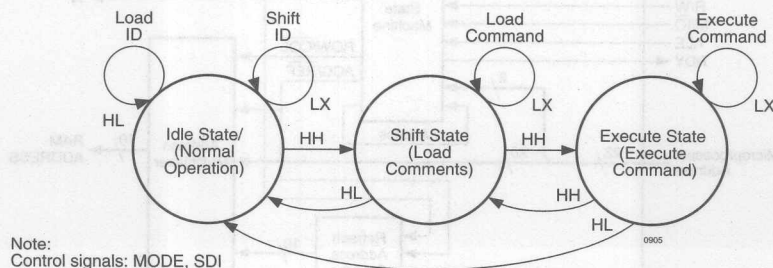
Implementing a basic DRAM timing control logic in the ispLSI 1032 takes up approximately 65% of the total logic available in the device. (It is with this in mind that the features needed for a specific design can be added to these basic logic blocks). With the ISP capability, many features can be added to accommodate the ever changing requirements of the system, microprocessor speeds, availability of DRAMs, and the memory configurations. Moreover, the changes are made only under the software control. Instead of having different production runs for various different options, the options are added at the in-system programming stage.

The programming of the ispLSI devices are handled via five TTL level interface signals. Of these five signals, four signals can be dual function, a programming function as well as an input during normal device operation. The ISP Enable (ispEN) signal is the one dedicated programming pin used to enable and disable the programming function. Once in programming mode, the mode control (MODE), serial data input (SDI), serial data clock (SCLK), and serial data output (SDO) signals control the entire programming process. The address and data required to program the device are serially shifted into the internal shift registers and the three state programming state machine steps through the programming sequence. The five-bit instructions within the state machine define all the necessary steps for programming. Figure 2 shows the ISP programming state machine with the control signal requirements for the state transitions. Refer to the ISP architecture and programming section of this handbook for a more detailed programming description.

Different System Speed

Designing with a different speed microprocessor requires a different DRAM timing controller. The

Figure 2. ISP State Machine



ispLSI Configurable Memory Controller

adjustments must be made in the state machine and refresh timer logic of the controller to account for the difference in speed. Without the capabilities of the ISP features, different boards with different PLD codes must be built to work with different processor speeds. By providing a simple programming circuitry on board to support the isp programming, the logic adjustments for different speed processor can be accomplished by in-system programming the different patterns via software control. Manufacture of these options are made simple and cost effective by not having to keep an inventory of prepatterned devices.

DRAM Feature Flexibility

DRAMs have many features from which the system designer can select. For the same DRAM configuration, the system designer can select from DRAMs that have different access schemes such as nibble mode, static column mode and page mode. Similarly, different memory refresh schemes can be chosen. The two choices of refresh schemes include the simple RAS only refresh and the option to perform hidden refresh with the CAS before RAS refresh scheme. Most of these various DRAM options can be supported by in-system programming the ispLSI devices. Again, the flexibility lies in the fact that the decision of what function the ispLSI will perform on board can be made after the decision has been made on which type of DRAMs are used on board.

Different DRAM Configuration

The ispLSI implementation of the DRAM timing controller makes the change of memory configuration very simple. Reprogramming of the address decoding and turning on the appropriate address strobe signals for different memory configuration can be done by in-system reconfiguration of the state machine and the address decoding of the ispLSI device. All of these changes can be accomplished under software control.

Memory Timing Controller Details

As shown in figure 1 the memory timing controller consists of four different logic blocks. The refresh timer, state machine, refresh address counter and memory address multiplexer. All boolean equations for the logic blocks are developed within the Lattice pDS Software. The entire memory timing controller design assumes that all the processor signals are typical of a commercially available processor with a clock speed of 25 MHz. DRAMs are arranged in four banks of 1M X 32-bit arrangement. All timing for the access and refresh sequences are shown in the timing diagram.

Refresh Timer

The function of the refresh timer is to generate a refresh request signal every 15.5 μ s. This refresh period is derived from the DRAM refresh requirement of 512 rows of refresh every 8 ms for the 1M X 1 DRAM. Based on the 25 MHz system clock frequency, the count value to divide the clock period to the refresh period is 200. Changing processor speed will only require a change of count value. Once the count value expires, the refresh timer generates an internal refresh signal to inform the state machine to perform a refresh cycle. When the state machine completes the refresh cycle, a refresh complete (RFC) signal is generated for the refresh timer. The refresh timer then resets the internal counter for the next refresh period.

ispLSI implementation of the refresh timer takes up three GLBs (A0-A2) within the device. The system clock is used to run the nine bit counter, RFC is the input signal to this block and REFRESH is the output signal of this logic block.

State Machine

The state machine can be further divided into four different sub-logic blocks. These sub-logic blocks consists of a RAS generator, CAS generator, 4-bit state machine which is divided into two state variable bits and two counter bits, and control signal generator. In the ispLSI 1032 implementation, the state machine logic block takes up 9 GLBs.

The 4-bit state machine is divided into a 2-bit state variable, named ST0 and ST1, and 2-bit state counter, named SCNT0 and SCNT1. The state diagram with its state transitions are shown in figure 3. In each of the access and refresh states, the state counter sequences through the operation until the sequence is complete. The purpose of the state variable bits are only to keep track of the state transitions. Once the state transition has occurred, the state counter bits take the responsibility of sequencing through the state.

The three states are divided as idle state, access state and refresh state. Based on the processor control signal and the internal refresh request signal, the state transition occurs from idle state to either access state or refresh state. If the refresh and access request happen at the same time, refresh request takes precedence over access request. When the refresh request is asserted during an access cycle, the refresh cycle follows right after the access cycle. The only other condition between the access and refresh request that the state machine needs to arbitrate is when the access request occurs

ispLSI Configurable Memory Controller

during the refresh sequence. The access feedback signal of the state machine is activated when the access request occurs during the refresh cycle. When the refresh cycle is complete, the access feedback signal is used to determine whether or not the access sequence needs to begin. The timing diagrams in figure 4 and 5 illustrate the control signal sequence for the access and refresh cycles, respectively.

Figure 3. DRAM Timing Controller State Machine

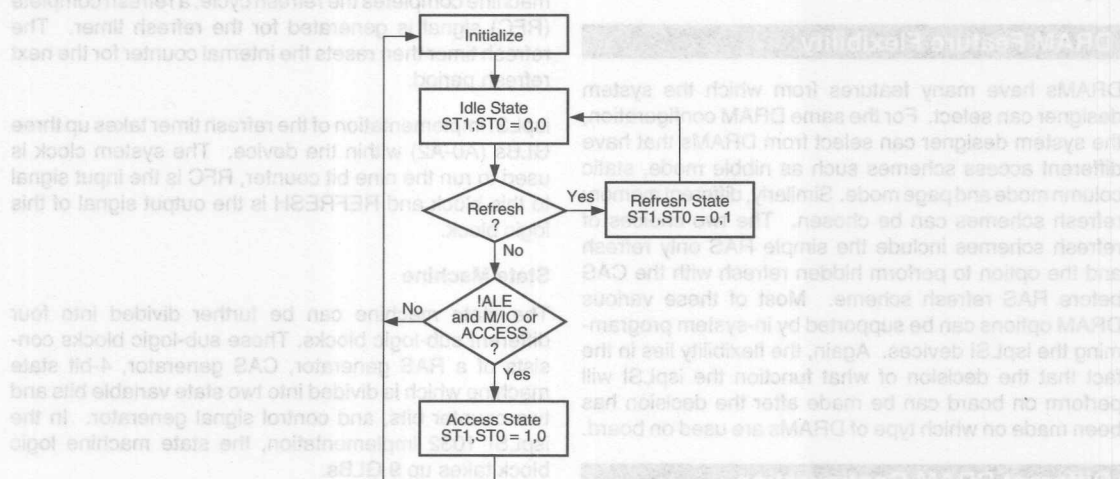
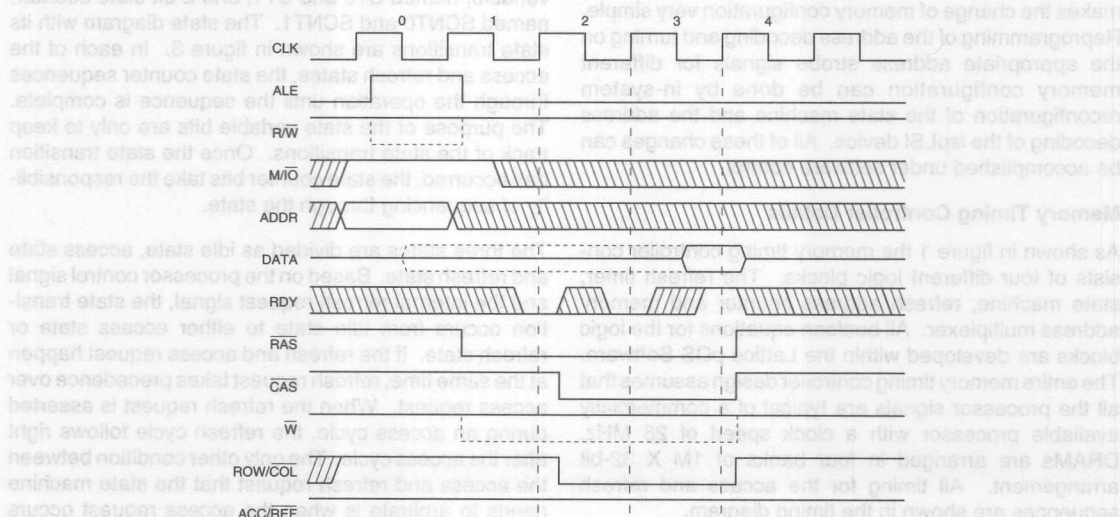


Figure 4. Access Cycle Timing



ispLSI Configurable Memory Controller

Refresh Address Counter

The refresh address counter keeps track of the rows of DRAM to be refreshed. This counter is only incremented on the falling edge of the RAS signal during refresh sequence. The ispLSI device implementation of this counter takes 3 GLBs.

Memory Address Multiplexer

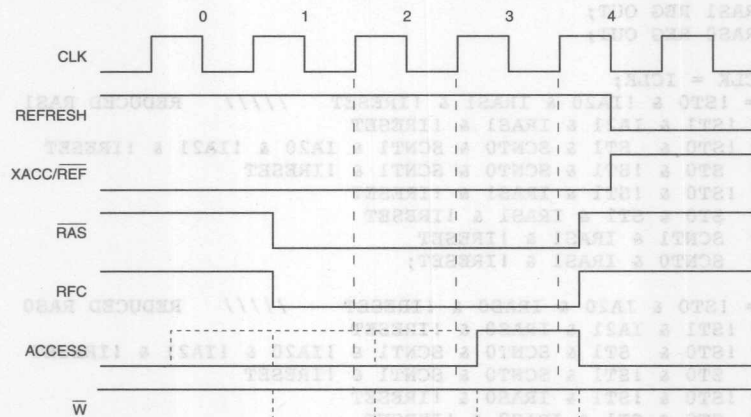
In access mode, determined by the ACC/REF internal signal, the memory address multiplexer multiplexes between the row and column address. Once in the refresh cycle, the refresh address comes from the refresh address counter. It takes 3 GLBs to implement the multiplexer in the ispLSI 1032.

Conclusion

The intention of this application section is to give an overview of how the ISP features can be used to improve the design features and the manufacturing process by using an example of a generalized DRAM timing controller. In addition, the software example given in the document should provide a good starting point for designers who need to implement a state machine based design. With the flexibility of the ispLSI devices the possibilities are limited only by one's imagination to implement innovative designs. The following sections list the Lattice Design file with the Boolean Equations and pinout for the ispLSI 1032.

4

Figure 5. Refresh Cycle Timing




```

//isp_app.ldf generated using Lattice pDS Version 2.50
LDF 1.00.00 DESIGNLDF;
DESIGN DRAM CONTROLLER 1.00;
PROJECTNAMEispAPPLICATIONS;
DESCRIPTION
DRAM CONTROLLER DESIGN FORispAPPLICATION.
IT INCLUDES FOUR MAJOR BLOCKS.
- REFRESH TIMER
- REFRESH ROW ADDRESS COUNTER
- ADDRESS MUX
- STATE MACHINE;

PART pLSI 1032-90LJ;

DECLARE
END; //DECLARE

SYM GLB C2 1 ;
///// ROW ADDRESS STROBE (RAS1,RAS0) GLB
SIGTYPE IRAS1 REG OUT;
SIGTYPE IRAS0 REG OUT;
EQUATIONS
  IRAS1.CLK = ICLK;
  IRAS1 = !ST0 & !IA20 & IRAS1 & !IRESET //REDUCED RAS1
  # !ST1 & IA21 & IRAS1 & !IRESET
  # !ST0 & ST1 & SCNT0 & SCNT1 & IA20 & !IA21 & !IRESET
  # ST0 & !ST1 & SCNT0 & SCNT1 & !IRESET
  # !ST0 & !ST1 & IRAS1 & !IRESET
  # ST0 & ST1 & IRAS1 & !IRESET
  # SCNT1 & IRAS1 & !IRESET
  # SCNT0 & IRAS1 & !IRESET;

  IRAS0 = !ST0 & IA20 & IRAS0 & !IRESET //REDUCED RAS0
  # !ST1 & IA21 & IRAS0 & !IRESET
  # !ST0 & ST1 & SCNT0 & SCNT1 & !IA20 & !IA21 & !IRESET
  # ST0 & !ST1 & SCNT0 & SCNT1 & !IRESET
  # !ST0 & !ST1 & IRAS0 & !IRESET
  # ST0 & ST1 & IRAS2 & !IRESET
  # SCNT1 & IRAS0 & !IRESET
  # SCNT0 & IRAS0 & !IRESET;

END
END;

SYM GLB A2 1 ;
///// REFRESH TIMER GLB2
SIGTYPE RQ8 REG OUT;
SIGTYPE RQ9 REG OUT;
SIGTYPE REFRESH REG OUT;
FJK11 (REFRESH,R_RATE,RFC,ICLK); //REFRESH REQUEST SIGNAL
EQUATIONS
  RQ8.CLK = ICLK;
  RQ8 = (RQ8 & !RFC)
  $$ (RQ7 & RQ6 & RQ5 & RQ4 & RQ3 & RQ2 & RQ1 & RQ0 & !RFC);
  RQ9 = (RQ9 & !RFC)
  $$ (RQ8 & RQ7 & RQ6 & RQ5 & RQ4 & RQ3 & RQ2 & RQ1 & RQ0 & !RFC);
  R_RATE = RQ7 & RQ6 & !RQ5 & !RQ4 & RQ3 & !RQ2 & !RQ1 & !RQ0;

END
END;

```

ispLSI Configurable Memory Controller

```

SYM GLB A1 1 ;
///// REFRESH TIMER GLB1
SIGTYPE RQ4 REG OUT;
SIGTYPE RQ5 REG OUT;
SIGTYPE RQ6 REG OUT;
SIGTYPE RQ7 REG OUT;
EQUATIONS
  RQ4.CLK = ICLK;
  RQ4 = (RQ4 & !RFC);
  $$ (RQ3 & RQ2 & RQ1 & RQ0 & !RFC);
  RQ5 = (RQ5 & !RFC);
  $$ (RQ4 & RQ3 & RQ2 & RQ1 & RQ0 & !RFC);
  RQ6 = (RQ6 & !RFC);
  $$ (RQ5 & RQ4 & RQ3 & RQ2 & RQ1 & RQ0 & !RFC);
  RQ7 = (RQ7 & !RFC);
  $$ (RQ6 & RQ5 & RQ4 & RQ3 & RQ2 & RQ1 & RQ0 & !RFC);
END
END;

SYM GLB A0 1 ;
///// REFRESH TIMER GLB0
SIGTYPE RQ0 REG OUT;
SIGTYPE RQ1 REG OUT;
SIGTYPE RQ2 REG OUT;
SIGTYPE RQ3 REG OUT;
EQUATIONS
  RQ0.CLK = ICLK;
  RQ0 = !RQ0 & !RFC;
  RQ1 = (RQ1 & !RFC);
  $$ (RQ0 & !RFC);
  RQ2 = (RQ2 & !RFC);
  $$ (RQ1 & RQ0 & !RFC);
  RQ3 = (RQ3 & !RFC);
  $$ (RQ2 & RQ1 & RQ0 & !RFC);
END
END;

SYM GLB D0 1 ;
///// ADDRESS MUX GLB0
SIGTYPE IRAM0 ASYNC OUT;
SIGTYPE IRAM1 ASYNC OUT;
SIGTYPE IRAM2 ASYNC OUT;
SIGTYPE IRAM3 ASYNC OUT;
EQUATIONS
  IRAM0 = ROW_COL & ACC_REF & IA0
  # !ROW_COL & ACC_REF & IA10
  # !ACC_REF & RCNTR0;
  IRAM1 = ROW_COL & ACC_REF & IA1
  # !ROW_COL & ACC_REF & IA11
  # !ACC_REF & RCNTR1;
  IRAM2 = ROW_COL & ACC_REF & IA2
  # !ROW_COL & ACC_REF & IA12
  # !ACC_REF & RCNTR2;
  IRAM3 = ROW_COL & ACC_REF & IA3
  # !ROW_COL & ACC_REF & IA13
  # !ACC_REF & RCNTR3;
END
END;

```

ispLSI Configurable Memory Controller

```

SYM GLB D1 1 ;
///// ADDRESS MUX GLB1
SIGTYPE IRAM4 ASYNC OUT;
SIGTYPE IRAM5 ASYNC OUT;
SIGTYPE IRAM6 ASYNC OUT;
SIGTYPE IRAM7 ASYNC OUT;
EQUATIONS
  IRAM4 = ROW_COL & ACC_REF & IA4      ///// ROW SELECT
  # !ROW_COL & ACC_REF & IA14          ///// COLUMN SELECT
  # !ACC_REF & RCNTR4;                  ///// REFRESH ADDR SELECT
  IRAM5 = ROW_COL & ACC_REF & IA5
  # !ROW_COL & ACC_REF & IA15
  # !ACC_REF & RCNTR5;
  IRAM6 = ROW_COL & ACC_REF & IA6
  # !ROW_COL & ACC_REF & IA16
  # !ACC_REF & RCNTR6;
  IRAM7 = ROW_COL & ACC_REF & IA7
  # !ROW_COL & ACC_REF & IA17
  # !ACC_REF & RCNTR7;

END
END;

SYM GLB D2 1 ;
///// ADDRESS MUX GLB2
SIGTYPE IRAM8 ASYNC OUT;
SIGTYPE IRAM9 ASYNC OUT;
EQUATIONS
  IRAM8 = ROW_COL & ACC_REF & IA8      ///// ROW SELECT
  # !ROW_COL & ACC_REF & IA18          ///// COLUMN SELECT
  # !ACC_REF & RCNTR8;                  ///// REFRESH ADDR SELECT
  IRAM9 = ROW_COL & ACC_REF & IA9
  # !ROW_COL & ACC_REF & IA19
  # !ACC_REF & RCNTR9;

END
END;

SYM GLB D5 1 ;
///// REFRESH ROW COUNTER GLB0
SIGTYPE RCNTR0 REG OUT;
SIGTYPE RCNTR1 REG OUT;
SIGTYPE RCNTR2 REG OUT;
SIGTYPE RCNTR3 REG OUT;
EQUATIONS
  RCNTR0.PTCLK = !IRAS0;              ///// USE RAS AS THE COUNTER CLOCK
  RCNTR0 = !RCNTR0 & !ACC_REF          ///// COUNT DURING REFRESH
  # RCNTR0 & ACC_REF;                  ///// HOLD DURING ACCESS
  RCNTR1 = (RCNTR1 & !ACC_REF)
  $$ ((RCNTR0 & !ACC_REF)
  # (RCNTR1 & ACC_REF));
  RCNTR2 = (RCNTR2 & !ACC_REF)
  $$ ((RCNTR1 & RCNTR0 & !ACC_REF)
  # (RCNTR2 & ACC_REF));
  RCNTR3 = (RCNTR3 & !ACC_REF)
  $$ ((RCNTR2 & RCNTR1 & RCNTR0 & !ACC_REF)
  # (RCNTR3 & ACC_REF));

END
END;

```

ispLSI Configurable Memory Controller

```

SYM GLB D6 1 ;
///// REFRESH ROW COUNTER GLB1
SIGTYPE RCNTR4 REG OUT;
SIGTYPE RCNTR5 REG OUT;
SIGTYPE RCNTR6 REG OUT;
SIGTYPE RCNTR7 REG OUT;
EQUATIONS
    ///// USE RAS AS THE COUNTER CLOCK
    RCNTR4.PTCLK = !IRAS0;
    RCNTR4 = (RCNTR4 & !ACC_REF)
    ///// COUNT DURING REFRESH
    $$ ((RCNTR3 & RCNTR2 & RCNTR1 & RCNTR0 & !ACC_REF)
    # (RCNTR4 & ACC_REF));
    ///// HOLD DURING ACCESS
    RCNTR5 = (RCNTR5 & !ACC_REF)
    $$ ((RCNTR4 & RCNTR3 & RCNTR2 & RCNTR1 & RCNTR0 & !ACC_REF)
    # (RCNTR5 & ACC_REF));
    RCNTR6 = (RCNTR6 & !ACC_REF)
    $$ ((RCNTR5 & RCNTR4 & RCNTR3 & RCNTR2 & RCNTR1 & RCNTR0 & !ACC_REF)
    # (RCNTR6 & ACC_REF));
    RCNTR7 = (RCNTR7 & !ACC_REF)
    $$ ((RCNTR6 & RCNTR5 & RCNTR4 & RCNTR3 & RCNTR2 & RCNTR1 & RCNTR0 &
    !ACC_REF)
    # (RCNTR7 & ACC_REF));
END
END;

SYM GLB D7 1 ;
///// REFRESH ROW COUNTER GLB2
SIGTYPE RCNTR8 REG OUT;
SIGTYPE RCNTR9 REG OUT;
EQUATIONS
    RCNTR8.PTCLK = !IRAS0;
    RCNTR8 = (RCNTR8 & !ACC_REF)
    $$ ((RCNTR7 & RCNTR6 & RCNTR5 & RCNTR4
    & RCNTR3 & RCNTR2 & RCNTR1 & RCNTR0 & !ACC_REF) # (RCNTR8 & ACC_REF));
    RCNTR9 = (RCNTR9 & !ACC_REF)
    $$ ((RCNTR8 & RCNTR7 & RCNTR6 & RCNTR5 & RCNTR4 & RCNTR3 & RCNTR2 &
    RCNTR1 & RCNTR0 & !ACC_REF)
    # (RCNTR9 & ACC_REF));
END
END;

SYM GLB C7 1 ;
///// STATE BITS GLB
SIGTYPE ST0 REG OUT;
SIGTYPE ST1 REG OUT;
FJK11 (ST0,JST0,KST0,ICLK);
FJK11 (ST1,JST1,KST1,ICLK);
EQUATIONS
    JST0 = !ST1 & !ST0 & REFRESH;
    KST0 = !ST1 & ST0 & SCNT1 & SCNT0;
    JST1 = !ST1 & !ST0 & !REFRESH & !IALE & IMIO_

```

ispLSI Configurable Memory Controller

```

      # !ST1 & !ST0 & !REFRESH & ACCESS;      // STATE BIT1 SET INPUT /
////
      KST1 = ST1 & !ST0 & SCNT1 & SCNT0
      # !ST1 & ST0 & SCNT1 & SCNT0;      // STATE BIT0 RESET INPUT /
////
      END
      END;

SYM GLB C6 1 ;
//// STATE COUNTER BITS GLB      //
      SIGTYPE SCNT0 REG OUT;
      SIGTYPE SCNT1 REG OUT;
      FJK11 (SCNT0,JSCNT0,KSCNT0,ICLK);
      FJK11 (SCNT1,JSCNT1,KSCNT1,ICLK);
      EQUATIONS
        JSCNT0 = !SCNT0 & ST1 & !ST0
        # !SCNT0 & !ST1 & ST0;      // STATE COUNTER BIT0 SET INPUT      //
        KSCNT0 = SCNT0 & ST1 & !ST0
        # SCNT0 & !ST1 & ST0
        # ST1 & !ST0 & SCNT1 & SCNT0
        # !ST1 & ST0 & SCNT1 & SCNT0;      // STATE COUNTER BIT0 RESET INPUT /
////
        JSCNT1 = !SCNT1 & SCNT0 & ST1 & !ST0
        # !SCNT1 & SCNT0 & !ST1 & ST0;      // STATE COUNTER BIT1 SET INPUT /
////
        KSCNT1 = SCNT1 & SCNT0 & ST1 & !ST0
        # SCNT1 & SCNT0 & !ST1 & ST0
        # ST1 & !ST0 & SCNT1 & SCNT0
        # !ST1 & ST0 & SCNT1 & SCNT0;      // STATE COUNTER BIT0 RESET INPUT /
////
      END
      END;

SYM GLB C5 1 ;
//// CONTROL SIGNALS GLB0      //
      SIGTYPE RFC REG OUT;
      SIGTYPE ACC_REF REG OUT;
      FJK11 (RFC,JRFC,KRFC,ICLK);
      FJK11 (ACC_REF,JACC_REF,KACC_REF,ICLK);
      EQUATIONS
        JRFC = !ST1 & ST0 & SCNT1 & !SCNT0;      // REFRESH COMPLETE SET INPUT
        //
        KRFC = !ST1 & ST0 & SCNT1 & SCNT0;      // REFRESH COMPLETE RESET INPUT //
        //
        JACC_REF = !ST1 & ST0 & SCNT1 & SCNT0
        # IRESET;      // ACCESS/REFRESH SET INPUT      //
        KACC_REF = !ST1 & !ST0 & REFRESH & !IRESET;      // ACCESS/REFRESH RESET INPUT
        //
      END
      END;

SYM GLB C1 1 ;
//// ROW ADDRESS STROBE (RAS3,RAS2) GLB      //
      SIGTYPE IRAS3 REG OUT;
      SIGTYPE IRAS2 REG OUT;
      EQUATIONS
        IRAS3 = !ST0 & !IA20 & IRAS3 & !IRESET      // REDUCED RAS3      //
        # !ST1 & !IA21 & IRAS3 & !IRESET
        # !ST0 & ST1 & SCNT0 & SCNT1 & IA20 & IA21 & !IRESET
        # ST0 & !ST1 & SCNT0 & SCNT1 & !IRESET

```


ispLSI Configurable Memory Controller

```

# !ST0 & !ST1 & IRAS3 & !IRESET
# ST0 & ST1 & IRAS3 & !IRESET
# SCNT1 & IRAS3 & !IRESET
# SCNT0 & IRAS3 & !IRESET;
IRAS3.CLK = ICLK;

IRAS2 = !ST0 & IA20 & IRAS2 & !IRESET      /////   REDUCED RAS2   /////
# !ST1 & !IA21 & IRAS2 & !IRESET
# !ST0 & ST1 & SCNT0 & SCNT1 & !IA20 & IA21 & !IRESET
# ST0 & !ST1 & SCNT0 & SCNT1 & !IRESET
# !ST0 & !ST1 & IRAS2 & !IRESET
# ST0 & ST1 & IRAS2 & !IRESET
# SCNT1 & IRAS2 & !IRESET
# SCNT0 & IRAS2 & !IRESET;
IRAS2.CLK = ICLK;

END
END;
SYM GLB B7 1 ;
///// COLUMN ADDRESS STROBE (CAS0,CAS1) GLB0   /////
SIGTYPE ICAS0 REG OUT;
SIGTYPE ICAS1 REG OUT;
FJK11 (ICAS0,JCAS0,KCAS0,ICLK);
FJK11 (ICAS1,JCAS1,KCAS1,ICLK);
EQUATIONS
///// CAS0 SET INPUT   /////
JCAS0 = ST1 & !ST0 & !IA1 & !IA0 & SCNT1 & SCNT0
# IRESET;
/////CAS0 RESET INPUT   /////
KCAS0 = ST1 & !ST0 & !IA1 & !IA0 & !SCNT1 & SCNT0 & !IRESET;
///// CAS1 SET INPUT   /////
JCAS1 = ST1 & !ST0 & !IA1 & IA0 & SCNT1 & SCNT0
# IRESET;
/////CAS1 RESET INPUT   /////
KCAS1 = ST1 & !ST0 & !IA1 & IA0 & !SCNT1 & SCNT0 & !IRESET;

END
END;

SYM GLB B6 1 ;
///// COLUMN ADDRESS STROBE (CAS2,CAS3) GLB1   /////
SIGTYPE ICAS2 REG OUT;
SIGTYPE ICAS3 REG OUT;
FJK11 (ICAS2,JCAS2,KCAS2,ICLK);
FJK11 (ICAS3,JCAS3,KCAS3,ICLK);
EQUATIONS
JCAS2 = ST1 & !ST0 & IA1 & !IA0 & !SCNT1 & SCNT0      /////   CAS2 SET INPUT
/////
# IRESET;
///// CAS2 RESET INPUT   /////
KCAS2 = ST1 & !ST0 & IA1 & !IA0 & SCNT1 & SCNT0 & !IRESET;
JCAS3 = ST1 & !ST0 & IA1 & IA0 & !SCNT1 & SCNT0///// CAS3 SET INPUT   /////
# IRESET;
///// CAS3 RESET INPUT   /////
KCAS3 = ST1 & !ST0 & IA1 & IA0 & SCNT1 & SCNT0 & !IRESET;

END
END;

```

ispLSI Configurable Memory Controller

```

SYM GLB B5 1 ;
///// CONTROL SIGNALS (ACCESS,WRITE) GLB1
SIGTYPE ACCESS REG OUT;
SIGTYPE IWREG REG OUT;
FJK11 (ACCESS,JACCESS,KACCESS,ICLK);
FJK11 (IWREG,JWREG,KWREG,ICLK);
EQUATIONS
    JACCESS = !IALE & IMIO_;      ///// MEMORY ACCESS REQUEST SET INPUT    ///
/
    KACCESS = ST1 & !ST0 & SCNT1 & SCNT0;/////MEMORY ACCESS REQUEST RESET
INPUT/////
    JWREG = !ACCESS & IRW_      ///// WRITE REGISTER SET INPUT    /////
    # ST1 & !ST0 & SCNT1 & SCNT0
    # IRESET;
    KWREG = !ACCESS & !IRW_ & !IRESET;    ///// WRITE REGISTER RESET INPUT
/////
END
END;

SYM GLB B4 1 ;
///// CONTROL SIGNALS (ROW/COL,RDY)GLB2
SIGTYPE ROW_COL REG OUT;
SIGTYPE IRDY REG OUT;
FJK11 (ROW_COL,JROW_COL,KROW_COL,ICLK);
FJK11 (IRDY,JRDY,KRDY,ICLK);
EQUATIONS
    JROW_COL = ST1 & !ST0 & SCNT1 & SCNT0///// ROW/COL SELECT SET INPUT    /////
    # IRESET;
    KROW_COL = ST1 & !ST0 & !SCNT1 & SCNT0 & !IRESET/////ROW/COL SELECT RESET SET
    INPUT/////
    JRDY = ST1 & !ST0 & SCNT1 & !SCNT0;    ///// READY SET INPUT    /////
    KRDY = ST1 & !ST0 & SCNT1 & SCNT0;    ///// READY RESET INPUT    /////
END
END;

SYM IOC IO16 1 ;
// ADDR 12 I/O CELL W/REG. INPUT //
XPIN IO XA12;
ID11 (IA12,XA12,IICLK);
END;

SYM IOC IO15 1 ;
// ADDR 11 I/O CELL W/REG. INPUT //
XPIN IO XA11;
ID11 (IA11,XA11,IICLK);
END;

SYM IOC IO14 1 ;
// ADDR 10 I/O CELL W/REG. INPUT //
XPIN IO XA10;
ID11 (IA10,XA10,IICLK);
END;

SYM IOC IO13 1 ;
// ADDR 9 I/O CELL W/REG. INPUT //
XPIN IO XA9;
ID11 (IA9,XA9,IICLK);
END;

```

ispLSI Configurable Memory Controller

```

SYM IOC IO12 1 ;
// ADDR 8 I/O CELL W/REG. INPUT //
XPIN IO XA8;
ID11 (IA8,XA8,IICLK);
END;

SYM IOC IO11 1 ;
// ADDR 7 I/O CELL W/REG. INPUT //
XPIN IO XA7;
ID11 (IA7,XA7,IICLK);
END;

SYM IOC IO10 1 ;
// ADDR 6 I/O CELL W/REG. INPUT //
XPIN IO XA6;
ID11 (IA6,XA6,IICLK);
END;

SYM IOC IO9 1 ;
// ADDR 5 I/O CELL W/REG. INPUT //
XPIN IO XA5;
ID11 (IA5,XA5,IICLK);
END;

SYM IOC IO8 1 ;
// ADDR 4 I/O CELL W/REG. INPUT //
XPIN IO XA4;
ID11 (IA4,XA4,IICLK);
END;

SYM IOC IO7 1 ;
// ADDR 3 I/O CELL W/REG. INPUT //
XPIN IO XA3;
ID11 (IA3,XA3,IICLK);
END;

SYM IOC Y2 1 ;
// INPUT REGISTER CLOCK (ALE) //
XPIN CLK XICLK;
IB11 (IICLK,XICLK);
END;

SYM IOC IO6 1 ;
// ADDR 2 I/O CELL W/REG. INPUT //
XPIN IO XA2;
ID11 (IA2,XA2,IICLK);
END;

SYM IOC IO5 1 ;
// ADDR 1 I/O CELL W/REG. INPUT //
XPIN IO XA1;
ID11 (IA1,XA1,IICLK);
END;

SYM IOC IO4 1 ;
// ADDR 0 I/O CELL W/REG. INPUT //
XPIN IO XA0;
ID11 (IA0,XA0,IICLK);
END;

```

```

SYM IOC IO3 1 ;
// READY I/O CELL, OUTPUT //
XPIN IO XRDY;
OB11 (XRDY,IRDY);
END;

SYM IOC IO2 1 ;
// ADDRESS LATCH ENABLE I/O CELL /
/
XPIN IO XALE;
IB11 (IALE,XALE);
END;

SYM IOC IO1 1 ;
// MEMORY OR I/O ACCESS //
XPIN IO XMIO_;
IB11 (IMIO_,XMIO_);
END;

SYM IOC IO0 1 ;
// READ WRITE SELECTION //
XPIN IO XRW_;
IB11 (IRW_,XRW_);
END;

SYM IOC Y0 1 ;
// SYSTEM CLOCK INPUT //
XPIN CLK XSYS_CLK LOCK 20;
IB11 (ICLK,XSYS_CLK);
END;

SYM IOC IO17 1 ;
// ADDR 13 I/O CELL W/REG. INPUT /
/
XPIN IO XA13;
ID11 (IA13,XA13,IICLK);
END;

SYM IOC IO18 1 ;
// ADDR 14 I/O CELL W/REG. INPUT /
/
XPIN IO XA14;
ID11 (IA14,XA14,IICLK);
END;

SYM IOC IO19 1 ;
// ADDR 15 I/O CELL W/REG. INPUT /
/
XPIN IO XA15;
ID11 (IA15,XA15,IICLK);
END;

SYM IOC IO20 1 ;
// ADDR 20 I/O CELL W/REG. INPUT //
XPIN IO XA20;
ID11 (IA20,XA20,IICLK);
END;

```

ispLSI Configurable Memory Controller

```
SYM IOC IO21 1;
// ADDR 21 I/O CELL W/REG.INPUT //
XPIN IO XA21
ID11 (IA21,XA21,IICLK);
END;
```

```
SYM IOC IO22 1;
XPIN IO XRESET;
IB11 (IRESET, XRESET);
END;
```

```
SYM IOC IO23 1;
XPIN IO XREFRESH;
IB11 (REFRESH, XREFRESH);
END;
```

```
SYM IOC IO24 1;
XPIN IO XRAM0;
OB11 (XRAM0, IRAM0);
END;
```

```
SYM IOC IO25 1;
XPIN IO XRAM1;
OB11 (XRAM1, IRAM1);
END;
```

```
SYM IOC IO26 1;
XPIN IO XRAM2;
OB11 (XRAM2, IRAM2);
END;
```

```
SYM IOC IO27 1;
XPIN IO XRAM3;
OB11 (XRAM3, IRAM3);
END;
```

```
SYM IOC IO28 1;
XPIN IO XRAM4;
OB11 (XRAM4, IRAM4);
END;
```

```
SYM IOC IO29 1;
XPIN IO XRAM5;
OB11 (XRAM5, IRAM5);
END;
```

```
SYM IOC IO30 1;
XPIN IO XRAM6;
OB11 (XRAM6, IRAM6);
END;
```

```
SYM IOC IO31 1;
XPIN IO XRAM7;
OB11 (XRAM7, IRAM7);
END;
```

```
SYM IOC IO32 1;
XPIN IO XRAM8;
OB11 (XRAM8, IRAM8);
END;
```

```
SYM IOC IO33 1;
XPIN IO XRAM9;
OB11 (XRAM9, IRAM9);
END;
```

```
SYM IOC IO34 1;
XPIN IO XST0;
OB11 (XST0, ST0);
END;
```

```
SYM IOC IO36 1;
XPIN IO XST1;
OB11 (XST1, ST1);
END;
```

```
SYM IOC IO38 1;
XPIN IO XSCNT0;
OB11 (XSCNT0, SCNT0);
END;
```

```
SYM IOC IO40 1;
XPIN IO XSCNT1;
OB11 (XSCNT1, SCNT1);
END;
```

```
SYM IOC IO41 1;
XPIN IO XACCESS;
OB11 (XACCESS, ACCESS);
END;
```

```
SYM IOC IO42 1;
XPIN IO XIWREG;
OB11 (XIWREG, IWREG);
END;
```

```
SYM IOC IO43 1;
XPIN IO XROW_COL;
OB11 (XROW_COL, ROW_COL);
END;
```

```
SYM IOC IO44 1;
XPIN IO XIRDY;
OB11 (XIRDY, IRDY);
END;
```

ispLSI Configurable Memory Controller

```
SYM IOC IO45 1 ;  
XPIN IO XRFC;  
OB11 (XRFC, RFC);  
END;
```

```
SYM IOC IO46 1 ;  
XPIN IO XACC_REF;  
OB11 (XACC_REF, ACC_REF);  
END;
```


Notes

```
END;  
SYN IOG IOAS I ;  
XPIN IO XREF;  
OBII (XREF, REC);  
END;  
SYN IOG IOAS I ;  
XPIN IO XACC_REF;  
OBII (XACC_REF, ACC_REF);  
END;
```

Introduction

The Universal Product Code was first implemented by the grocery industry in 1973 as a method of improving inventory control and checkout times. As its benefits were realized, it soon spread throughout the retail industry. UPC Version A is a simple numeric only code encoding a 12 digit number into a continuous, fixed length symbol. UPC Version E is similar to Version A except that it only encodes 6 digits.

Figure 1. UPC Version A - 12 Digit Code

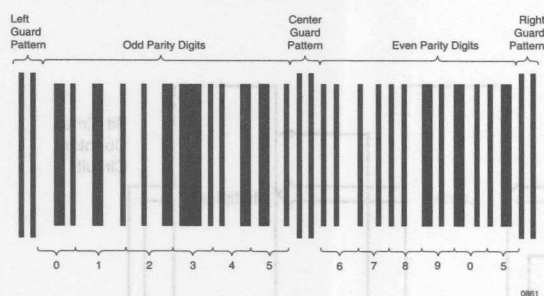
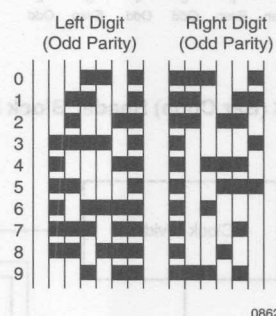


Figure 1 shows an example Version A pattern which encodes the number sequence 01234567890. The Version A pattern is divided into two halves. Each half consists of a guard pattern and six digits, with a guard pattern separating them. The patterns used to encode the digits on the left half have an odd number of bits (Odd Parity), and the patterns used to encode the digits on the right half have an even number of bits (Even Parity). This allows error checking to be performed, and a symbol which has been scanned backwards can be detected by detecting even parity codes being received before odd parity codes.

The basic width of a bar or space is determined by the width of the guard bars. This is defined to be in the range of 10.4 to 26 mils wide. Bar and space widths can be anywhere from 1 to 4 guard bar widths wide. The guard bars are usually printed with a slightly longer length than the data bars to allow a greater scanning tilt angle.

As mentioned before, the code which is used to represent numbers on the left side of the code is different from the code used on the right half. The data is encoded as a 2 of 7 Code using two bars and two spaces to describe 20 unique patterns. These 20 patterns encode the 10 numbers with both odd and even parity. The patterns are shown in Figure 2.

Figure 2. UPC Encodation Patterns

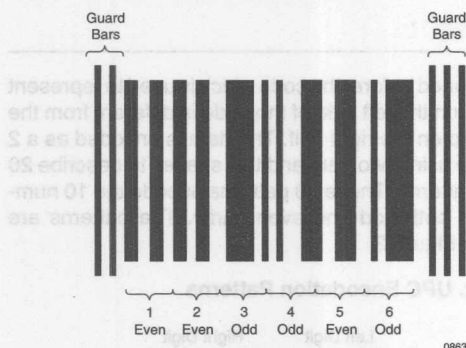


The first digit of the twelve is called the number system digit and is used to indicate the type of product the code is identifying. The next five digits are the Manufacturers ID Number as assigned by the Uniform Code Council, and the next five digits are the Item ID number assigned by the manufacturer. The last digit is a check digit. The value of this digit is based on the weighted sum of all of the other digits in the number. Using a weighted sum allows checking for transposition errors to be performed if the number is manually entered.

Figure 3 shows an example UPC Version E symbol which encodes the digits 123456. This code was specified to label small items. Because many of the digits in the Manufacturers ID Number and Item ID Number are frequently zeros, by suppressing these zeros using a standard compression process the number of digits can be reduced from 12 to 5. The last digit in the symbol indicates the type of suppression used in defining the symbol. This pattern uses intermixed digits of odd and even parity using the same encoding patterns shown in Figure 1.

Bar Code Reader

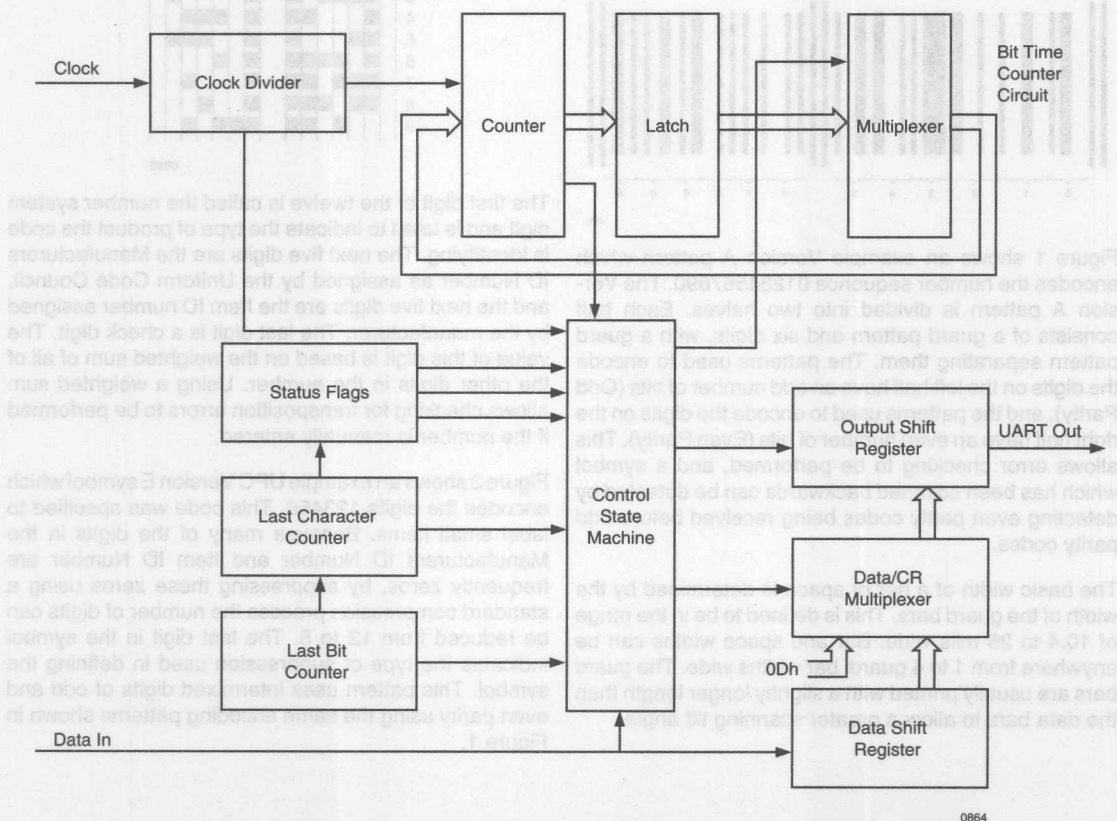
Figure 3. UPC Version E - 6 Digit Code



This example design shows how to use a Lattice pLSI device to implement a standard Universal Product Code (UPC), or Bar Code reader in a single chip. The chip that will be designed will receive digital signals from a Bar Code wand, determine the timing of those signals, separate the data from the wavetrain, and transmit that data asynchronously to a P.C. RS232 serial port. Both UPC Version A (12 Digit) and UPC version E (6 Digit) types will be decoded.

The main components of the block diagram in Figure 4 are described in the Theory of Operation section.

Figure 4. UPC (Bar Code) Reader Block Diagram



Theory of Operation

The UPC, or Bar Code, Reader consists of six major parts, which are the Clock Divider, Bit Time Counter, Data Shift Register, Control State Machine, Status Bits and UART. A complete description of each component follows:

Clock Divider

The Clock Divider Circuit (shown in Figure 8) can be divided into three parts. The divider takes the 1.8432 MHz reference clock and first divides it by 16 using a four bit counter CBU44 to generate 115.2 KHz. This is used as the main reader frequency, and was chosen primarily for two reasons. First, a frequency was needed that was fast enough so the clock skew at the edges of the received wand data is minimal in relation to Terminal Count (when the data is valid). Clock skew is caused by the data edge not being aligned with the rising edge of the clock. They can be, in the worse case, a complete clock period apart. If, for example, the data period is very small and the clock frequency is very slow, the width of the first guard bar will correspond to only a few counts. Again under worse case conditions, the clock skew at the starting edge of data added to the clock skew at the end of data can cause the data to be either sampled twice, or not at all. This condition is worsened if the data periods are varying. Secondly, the frequency had to be slow enough so that a nine bit counter would be sufficient to determine the width of the guard bar. Thus 115.2 KHz was selected as the reader frequency.

The 115.2 KHz is further divided using a two bit counter CBU42. By preloading one and counting up, the end result is 38.4 KHz. In the final stage, 38.4 KHz is divided by four using CBU22 to generate 9.6 KHz, which is used in the transfer of data over the UART.

BIT TIME COUNTER

The Bit Time Counter Circuit (shown in Figure 9) consists of two counters, storage flip-flops and logic gates. The two counters namely CBUD8 and CBUD1 form the nine bit counter which is used in determining the width of the first guard bar in the code. Once the leading edge of the first guard bar is received, the counters collectively start counting down. At the end of the first guard bar, the counters will contain a value relative to the width of the guard bar. For the most reliable reading, the data should be sampled in the middle of a bar or space. To achieve this, the value corresponding to the relative width of the first guard bar is divided by two and stored in the three

sets of flip-flops namely two FD24s and one FD21. Thus, for each subsequent bar or space, the shifted value is preloaded into the counters and the counters are allowed to count up to terminal count. Terminal count is then used to strobe the input data into the data shift register. The terminal count circuitry which determines when the data becomes valid consists of two parts, namely the edge detector and the CNTTC detector. The edge detector generates a pulse when the data makes a transition from either low to high or high to low and reloads the nine bit counter with the stored width value. The duration of the pulse is one clock period. The edge pulse also resets the toggle flip-flop which is part of the CNTTC detector circuitry. Thus when the nine bit counter reaches zero and the Sample signal goes high, the CNTTC flip-flop is set, which enables the data shift register, and data is latched. If an edge is not detected, the toggle flip-flop lets the counter count the stored guard bar width twice before CNTTC flip-flop is set and data is taken. The actual pDS code for this portion of the Bar Code Reader can be found in Figure 17.

The reason behind having the edge detector circuitry is to align the CNTTC pulses with the center of the data (bars and spaces). Since it is extremely hard to move the wand at a constant rate, the data pulses tend to have varying periods. If absolutely no alignment is done, the CNTTC pulses become invalid once the accumulated change in the data pulses is greater than half the width of the guard bar. Thus by starting the nine bit counter at the edges of the data, the above mentioned problem is greatly reduced. However, the problem still remains if the data periods reduce or enlarge more than half the width of the first guard bar. Therefore, it is recommended to move the wand as steadily as possible to keep the data pulses within a small margin, roughly half the width of the first guard bar.

DATA SHIFT REGISTER

The Data Shift Register (shown in Figure 12) consists of two 4-bit shift registers (SRR24). The Data Shift Register is used to store the incoming data until a complete character has been received. As discussed in the description of the Universal Product Code section, the bar code is 2 of 7 code. Thus, a complete character is received after every 7 bits. Moreover, Version E and Version A codes are 6 and 12 characters long respectively. Thus two counters are needed to keep track of the bit and the character counts. Although not directly part of the Data Shift Register, two CBU34 counters keep track of the number of bits and characters received. Since CBU34s are 4-bit counters, they are preloaded with nine

Bar Code Reader

for the bit count and ten for the character count respectively. The character counter, when reading the Version A case, is reset after the sixth character to accommodate the 12 characters in the Bar Code. After counting the seven bits, the bit counter generates LASTBIT signal which tells the rest of the logic a complete character has been received. The character counter, on the other hand, after receiving six characters asserts the LASTCHAR signal. The LASTCHAR signal is used by the Control State Machine to define various machine states.

Figure 5. State Diagram

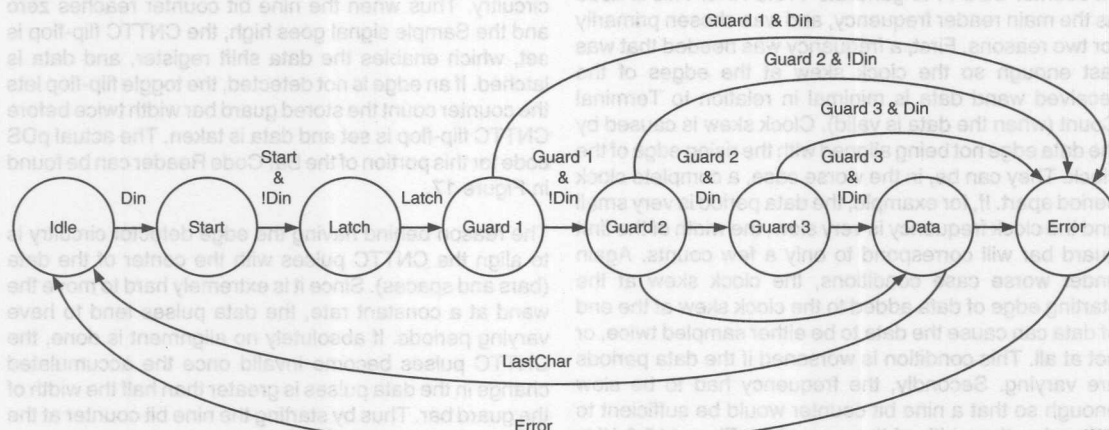
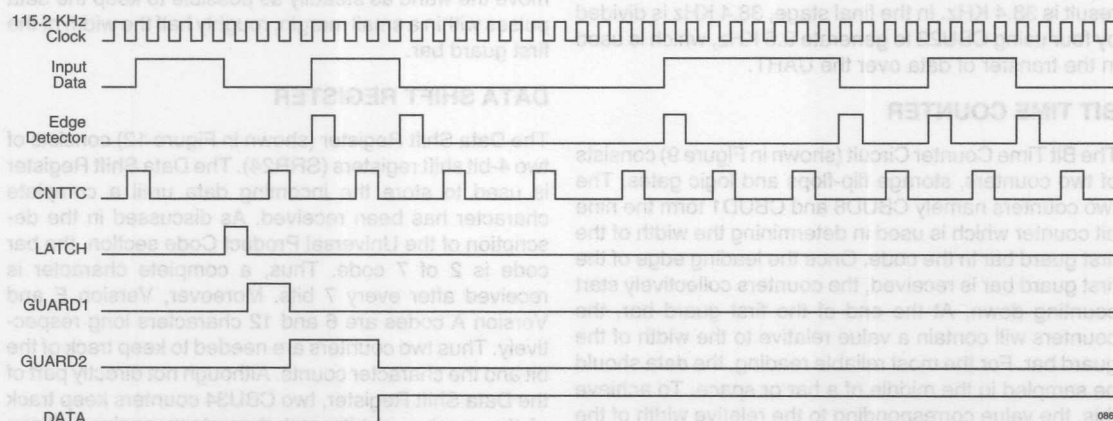


Figure 6. Data Waveforms Showing Machine States



CONTROL STATE MACHINE

The Control State Machine (shown in figure 14 and 15) is the heart of the design. It is based on three state determining variables namely SB0, SB1 and SB2. The job of the state machine is to detect and generate the basic width timing signals, generate the control signals for the Data Shift register and detect errors. The state diagram is shown in figure 5. Figure 6 shows a typical wave form highlighting the various machine states. A complete description of each state is as follows:

The first state is the IDLE state. In this state, the Reader is waiting for a low to high data transition. This corresponds to the wand and seeing the first guard bar signifying the start of the Bar Code. IDLE is reached after 6 consecutive zeros are seen in the input data stream signifying the end of the Bar Code. IDLE is also reached immediately after an error or after the end of the first half of a Version A pattern.

Once a low to high transition is detected, the Reader goes into the START state. As figure 5 shows, this is initiated by a DIN value which means the data received was a logic high. Another point to note is START can only be reached from the IDLE state. Once in the START state, the Bit Time Counter continuously decrements from zero. This state continues until the end of the first guard bar.

The end of the first guard bar is detected when a high to low data transition takes place. This places the Reader in the LATCH state, which is only a single 115.2 KHz bit wide. LATCH disables the Bit Time Counter and loads the counted value into the storage flip-flops. Remember that the value stored is actually one half the counted value. This is done to shift the sampling point to the middle of the bit period. LATCH also loads the stored value in the flip-flops into the Bit Time Counter.

Once the value is loaded in the Bit Time Counter, the Reader enters the GUARD1 state. In this state, the Bit Time Counter is again enabled but instead of counting down, the counter counts up. The Edge Detector circuit is also enabled. Once the Bit Time Counter reaches zero, the Sample signal goes high. This sets the CNTTC flip-flop which enables the shift register and data is read, and should be logic low. If it is, the Reader goes into the GUARD2 state. Otherwise, IDLE state is reached following the ERROR state. Before entering the GUARD2 state but after CNTTC, the stored bit width value is again loaded into the counter.

As figure 5 shows, the GUARD2 state is reached when the Reader is in the GUARD1 state and a !DIN is received. In the GUARD2 state, the Bit Time Counter is again enabled and allowed to count up. However, the CNTTC flip-flop is disabled so the next Sample does not force the Shift Register to take data. This is achieved through the toggle flip-flop which is part of the CNTTC circuitry. Thus when the Bit Time Counter counts up to zero, the bit width value is again loaded into the counter and is allowed to count up. However, the Sample signal does reset the toggle flip-flop so the next Sample signal sets the CNTTC flip-flop which in turn enables the Shift Register and data is taken in. Remember the Edge Detector circuitry was enabled in the GUARD1 state.

Thus what really happens is after the first Sample is detected in the GUARD2 state, the Edge Detector circuitry also notices the low to high transition of data and loads the stored bit width into the counter and resets the CNTTC flip-flop. Thus the GUARD2 state is the first time data alignment takes place. Once the Bit Time counter counts up to zero and Sample is enabled, data is taken in. If the data is logic high, the control circuitry checks if LASTHALF status bit is high. If it is, then the Reader goes into the GUARD3 state. Otherwise, the Reader enters the DATA state. If instead of a logic high, a logic low is received, the Reader goes to the IDLE state following the ERROR state.

In the GUARD3 state, the same sequence of steps are repeated as in the GUARD2 state except that !DIN is the correct data type used to bring the Reader in the DATA state. Also, the Edge Detector circuitry is in effect and influences the CNTTC flip-flop. One important point to note is the GUARD3 state is entered only if the LASTHALF status bit is high. The GUARD3 is the extra guard bit found in the center guard pattern of a Version A code. The GUARD3 state looks for that bit to be low at the proper time to enter the DATA state and goes to the ERROR state if it is not.

In the DATA state, the Reader is ready to receive data bits, assemble them into 7 bit data words and transfer them out over the UART. In the DATA state, Edge Detector circuitry is again enabled and, depending on the edge transitions, Bit Time Counter alignment with input data takes place. If an edge is not detected, the toggle flip-flop ensures proper operation of the Reader. Once six characters are read, the Reader goes into the IDLE state. The IDLE state can also be reached if six consecutive zeros are read as discussed in the IDLE state description above.

The ERROR state is reached if in either of the GUARD bits the correct data type is not read in. Thus the ERROR state indicates an abnormal condition has been detected. When this happens, the Reader sets the ERROR flag and reverts to the IDLE state and awaits the start of a new character.

Table 1 provides a list of all the states, including a short description of each state and a list of conditions which cause that particular state to occur.

Bar Code Reader

Table 1. State Table

S 2	S 1	S 0	Name	Equation	Description
0	0	0	IDLE	DEFAULT	Waiting for the first guard bit to appear
0	0	1	START	DIN&IDLE	Bit width counter is decremented
0	1	0	LATCH	START&IDIN	Counted width stored and loaded back into the bit width counter
0	1	1	GUARD1	LATCH	Verifies a space was seen
1	0	0	GUARD2	GUARD1 & IDIN	Verifies a bar was seen
1	0	1	GUARD3	GUARD2 & LASTHALF & DIN	Verifies a space was seen
1	1	0	DATA	GUARD2 & DIN & !LASTHALF # GUARD3 & IDIN & LASTHALF	Device ready to receive data
1	1	1	ERROR	GUARD1 & DIN & CNTTC # GUARD2 & IDIN & CNTTC # GUARD3 & DIN & CNTTC	Abnormal condition has been detected

STATUS BITS

Four Status Bits are used (shown in figure 12 and 13) to keep track of the progress of the read. LASTCHAR is used to store the fact that six characters have been read. This flip-flop is reset when the Reader is in the IDLE state. ERRORL stores the fact an error occurred while reading the code. As mentioned before, the ERROR state occurs when in either of the GUARD bits, the correct data type is not read. Moreover, since a seven bit data word is read and an eight bit data transfer word is used, the ERRORL bit is transmitted as the most significant bit of each data word transferred out. Thus the data receiving device on the other end of the UART can check the eighth bit and determine if any errors had occurred while reading the Bar Code. The ERRORL flip-flop is also reset when the Reader is in the IDLE state. The LASTHALF bit keeps track of which part of the Version A code is being scanned. Since the two halves of the Version A code are decoded independently (the reader goes into the IDLE state after the first six characters are read), the logic needs to keep track of which half is being processed. Thus when the character counter counts the first six characters, the LCHAR signal goes high which in turn sets the LASTHALF flip-flop. The LASTHALF bit is used in decoding the extra GUARD bit present in the center guard pattern which is the beginning of the second half of the code. The logic knows if the LASTHALF bit is low, the DATA state follows the GUARD2 state. Otherwise, the GUARD3 state follows the GUARD2 state. The DATA state comes after the GUARD3 state. Reset of the LASTHALF flip-flop is either caused by reading six consecutive zeros or when complete twelve characters of the

Version A code are read. Finally, DATAREADY is used to gate the transfer of a data word from the Data Shift Register to the Output Shift Register. From the Output Shift Register, data is transferred over the UART. DATAREADY flip-flop is set every time seven bits are read. Reset is caused by the DATTAKEN signal.

NOTE: LASTHALF sets and resets another flip-flop called SECOND_HALF shown in Figure 12. SECOND_HALF is only used in decoding the last bit of the twelfth character in Version A code. The reason for SECOND_HALF is because the last bit of even parity digits is always logic low. When the last bit is read, the Reader immediately goes into the IDLE state without transferring the correct byte out because as soon as LCHAR is enabled, the LASTHALF flip-flop is reset. This takes the State Machine out of the DATA state and into the IDLE state. By having the State Machine depend on SECOND_HALF rather than LASTHALF for the twelfth character, the problem is resolved.

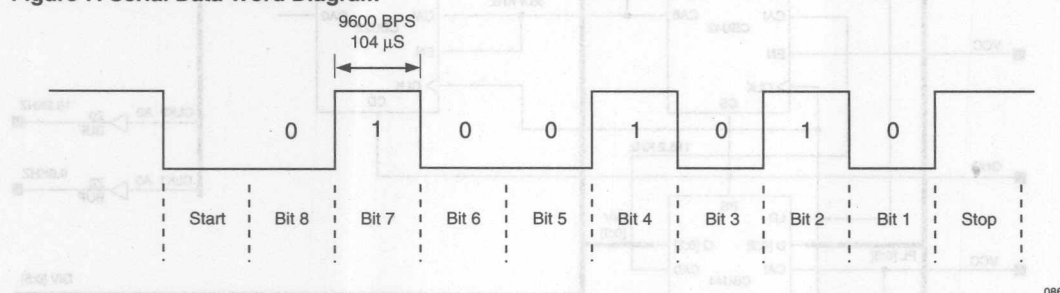
UART

The UART circuitry (shown in Figure 16) consists primarily of a shift register SRR31. Also shown on the schematic are two other shift registers and four sets of multiplexers. All these components are used in transferring the data out properly. The multiplexers, namely MUX22, select between data from the Data Shift Register and a hard coded 0D Hex (ASCII Carriage Return). The carriage return is sent whenever six consecutive ones are re-

ceived. It informs the receiver on the other end of the UART that the Bar Code has been read. The shift register SRR38 is the Output Shift Register. The data which accumulated in the Data Shift Register is transferred to the Output Shift Register once the DATTAKEN signal goes high which is driven by the DATAREADY signal. In

the Output Shift Register, using the second shift register SRR31 for clocking purposes, data is converted into the standard RS232 format and shifted into the UART Shift Register. The UART supports TTL signals. Thus a level shifter is needed at the output of the UART shift register. Figure 7 shows the data format out of the UART Shift Register.

Figure 7. Serial Data Word Diagram



HINTS ON TRANSLATING BAR CODE DATA TO ASCII

The Lattice pLSI device receives data from the wand and transfers it out through the UART Register. The UART Register supports TTL logic levels. An RS232 level shifter is needed before the data can be properly processed by the PC. The data format is: 8 Data bits, No parity, 1 Stop bit. The data rate is 9600 BAUD. The PC's serial port has to be configured accordingly. As mentioned in the Universal Product Code description, the data words either have even parity or odd parity. Thus a look up table is needed so that the received data words can be converted to their ASCII equivalent. Each type of parity has ten different codes for the ten digits. Since the wand can be scanned either from left to right or from right to left, the look up table has to have forty entries. Once the look up table is implemented, the received data can be compared and the proper ASCII code can be printed out. The look up table is as follows:

LEFT TO RIGHT SCANNING:

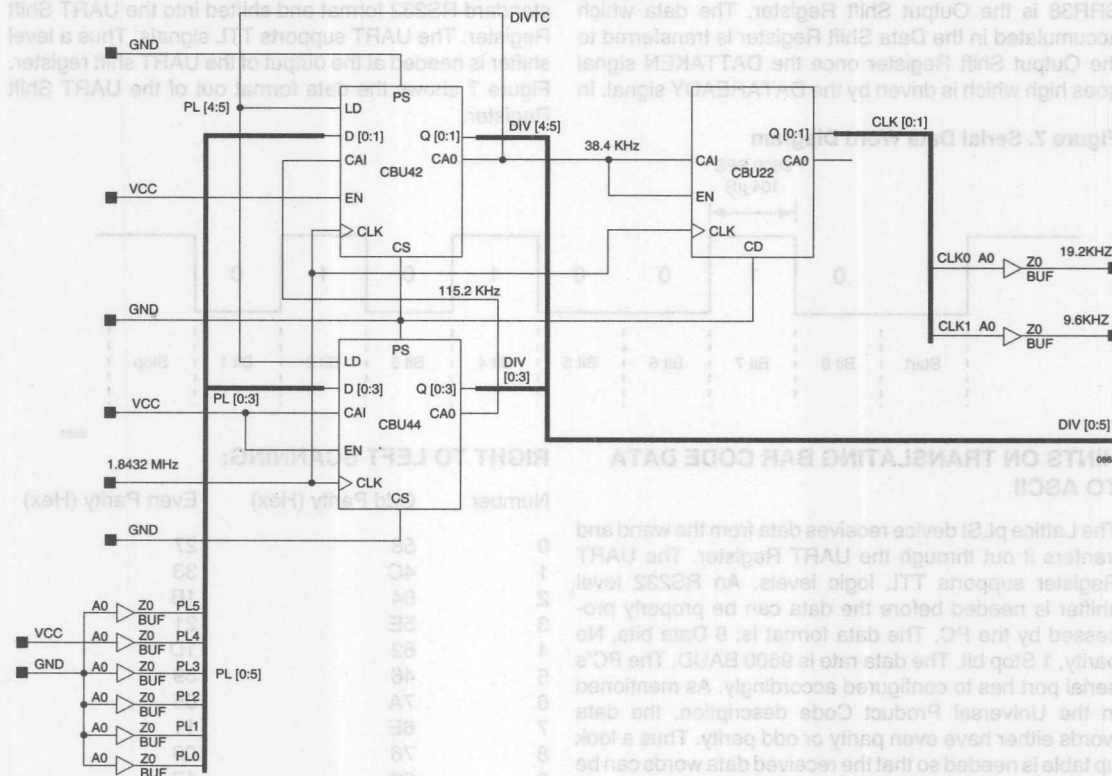
Number	Odd Parity (Hex)	Even Parity (Hex)
0	0D	72
1	19	66
2	13	6C
3	3D	42
4	23	5C
5	31	4E
6	2F	50
7	3B	44
8	37	48
9	0B	74

RIGHT TO LEFT SCANNING:

Number	Odd Parity (Hex)	Even Parity (Hex)
0	58	27
1	4C	33
2	64	1B
3	5E	21
4	62	1D
5	46	39
6	7A	05
7	6E	11
8	76	09
9	68	17

Bar Code Reader

Figure 8. Clock Divider





Bar Code Reader

Figure 10. CNTTC Detector Circuitry

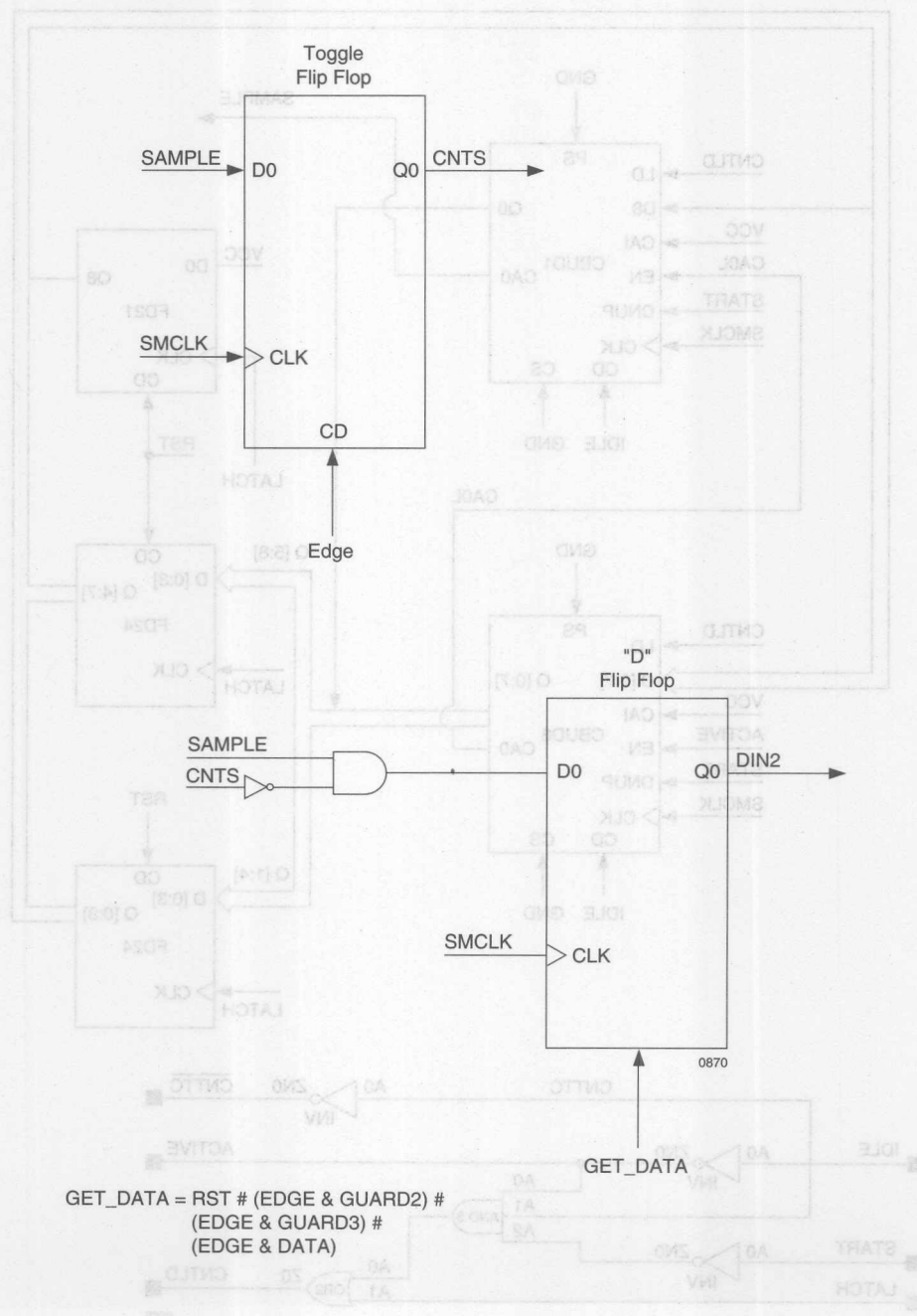
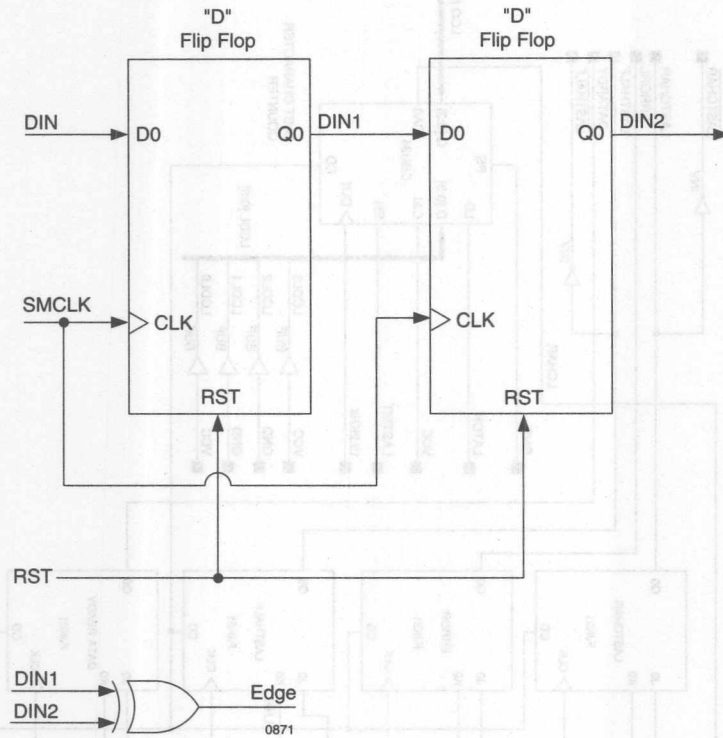


Figure 11. Edge Detector Circuitry



Bar Code Reader

Figure 12. Data Shift Register and Status Bits

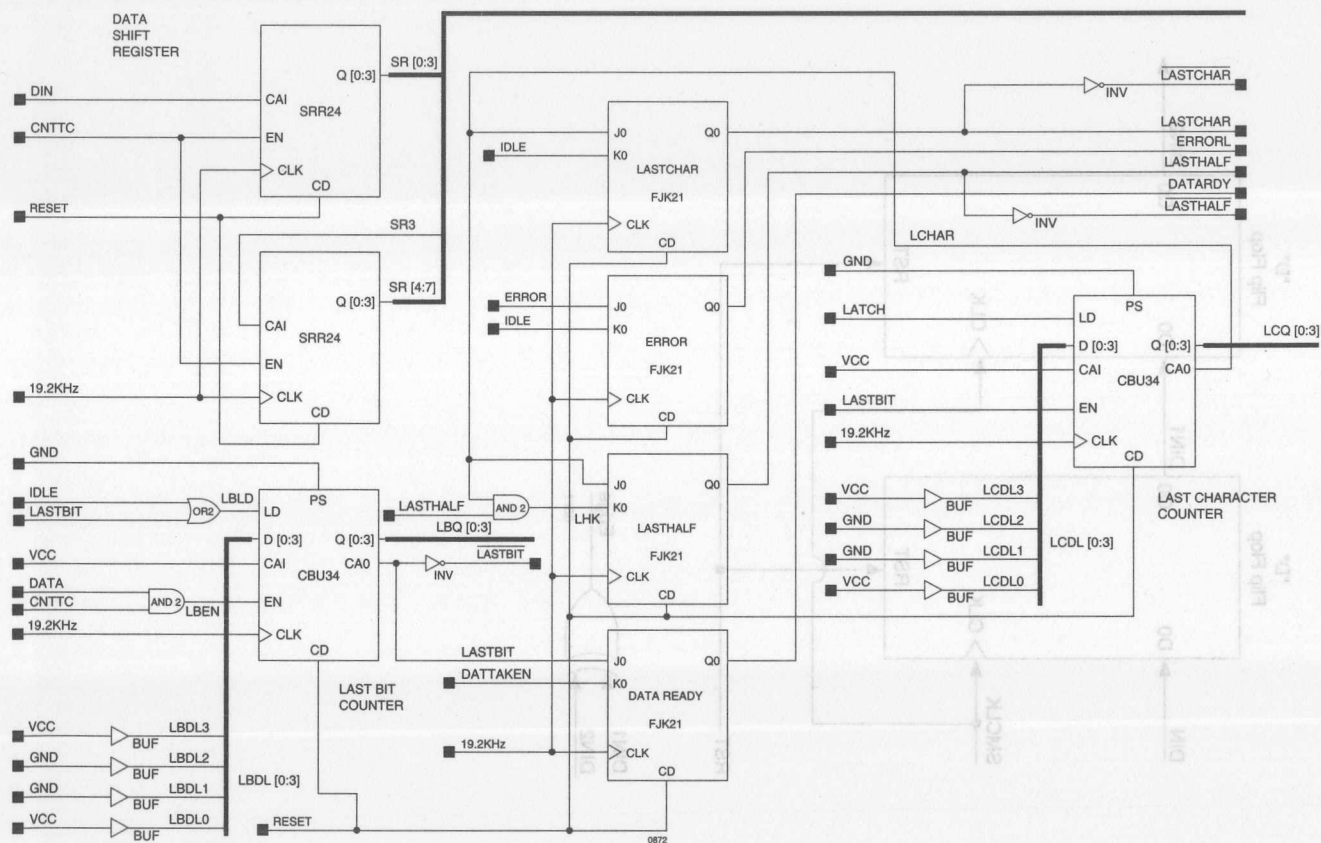
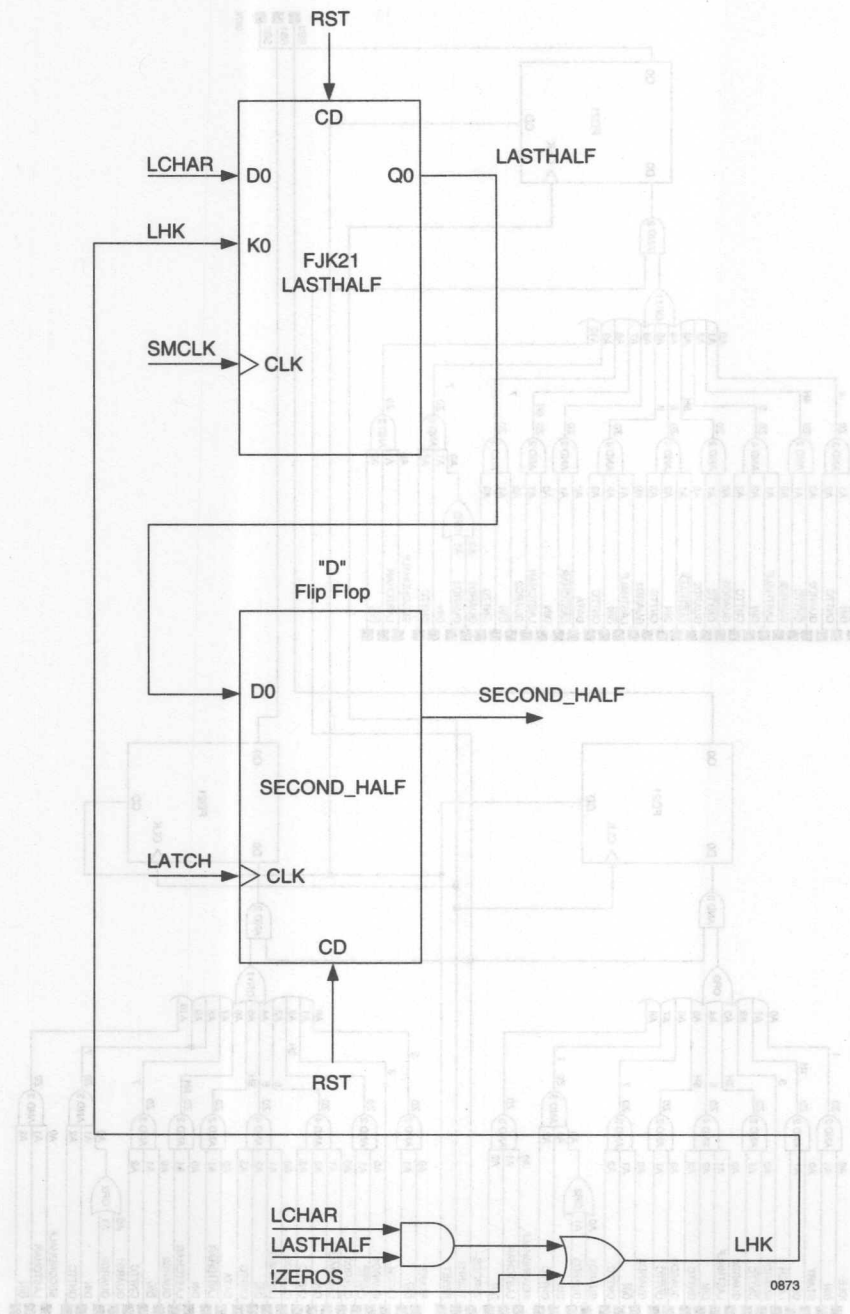


Figure 13. Second Half Circuit



Bar Code Reader

Figure 14. State Machine Registers

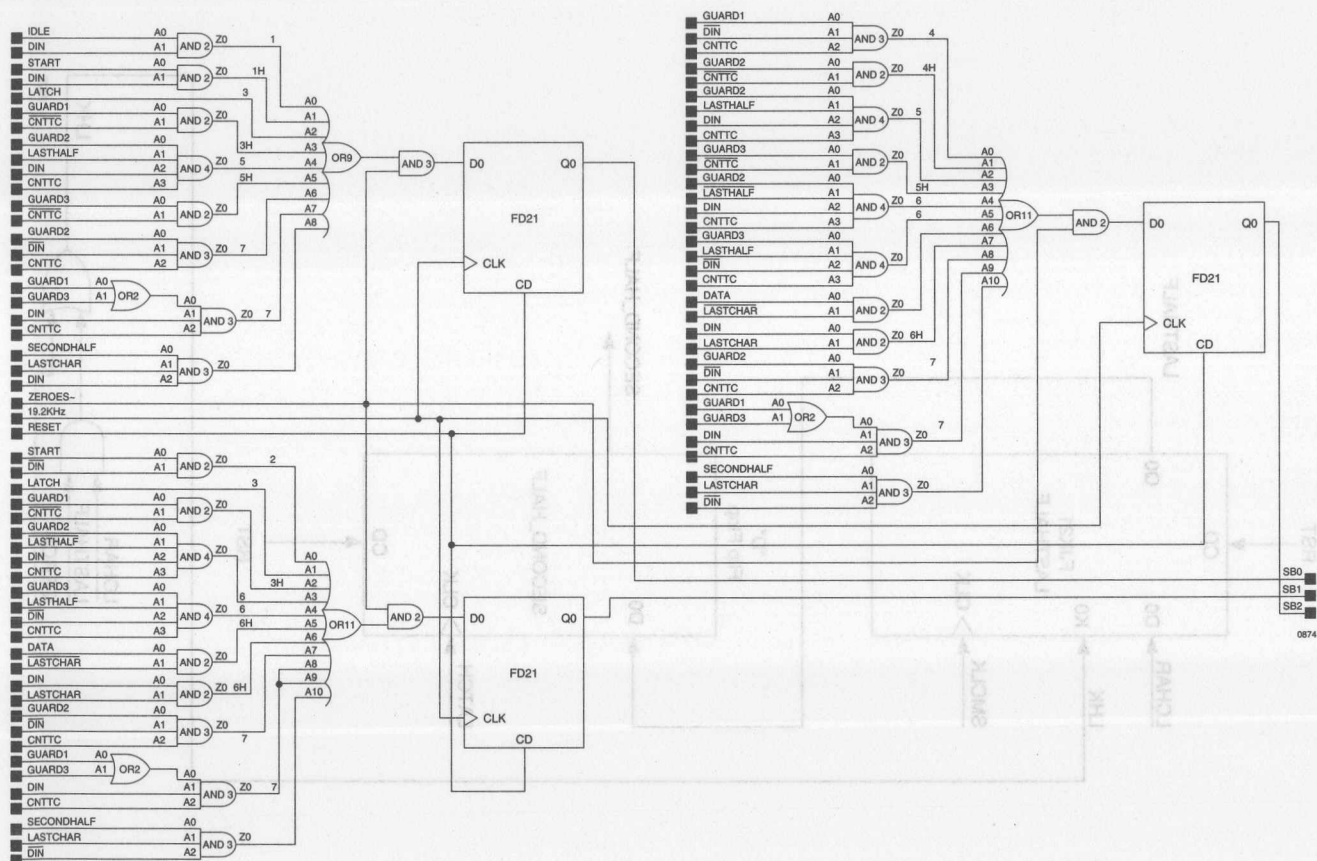
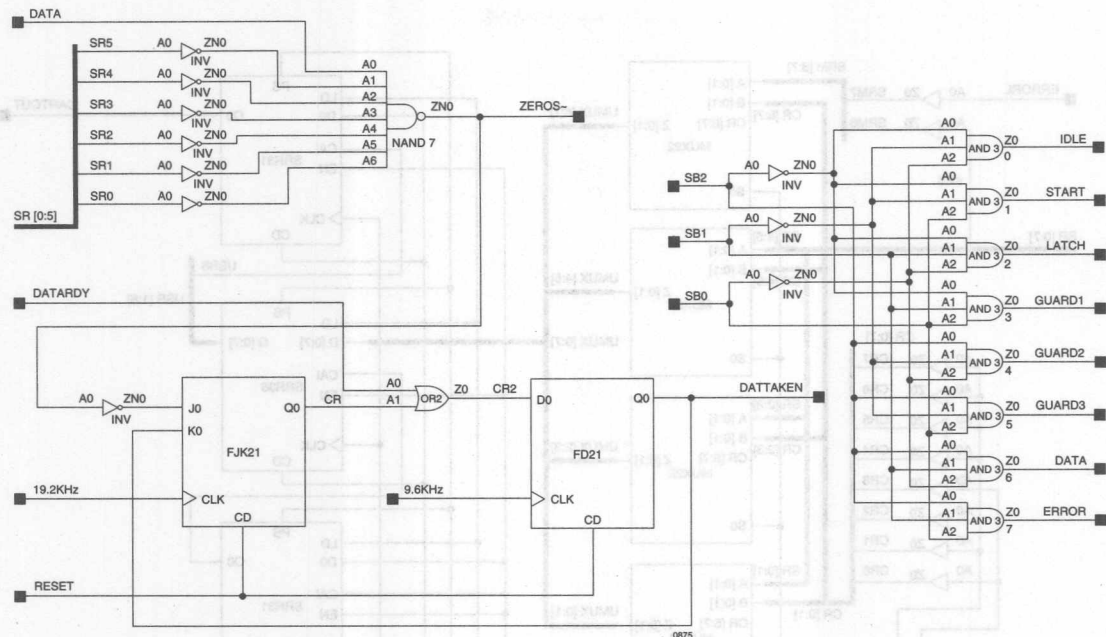


Figure 15. State Machine Decode



Bar Code Reader

Figure 16. Output Shift Register

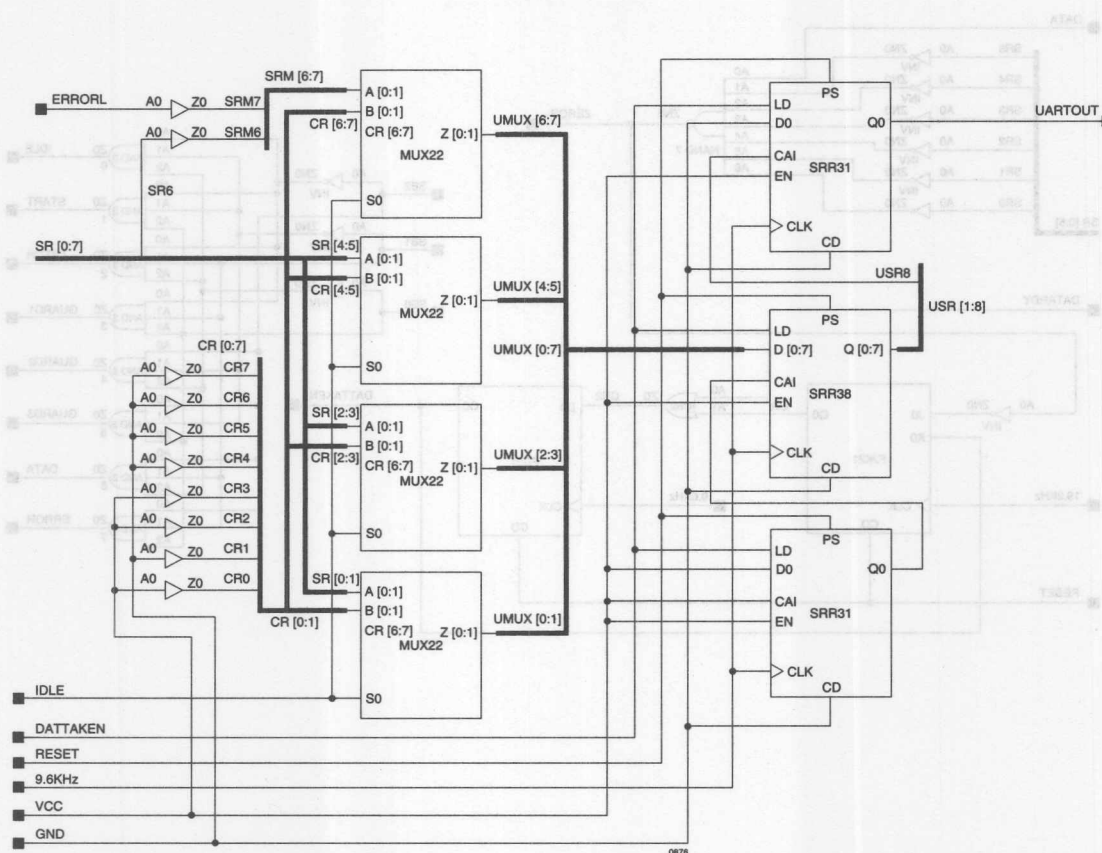


Figure 17. Portion of .LDF file

```
SYM GLB A2 1;
//      8-bit up/down counter wih Async preset, parallel load,
//      enable, up/dn, Async and Sync clear.  Uses 3 GLBs
//      Used as part of 9-bit counter to store the first bar's width value.
//      Bit 9 is located in GLB A5.
CBUD8 ([C0..C7], CNTTC, [L0..L7], VCC, SMCLK, GND, CNTLD, ACTIVE, START, IDLE, GND)];
END;

SYM GLB A3 1;
//      This is bit 9 of the 9-bit value storing the first bar's width
CBUD1 (8, SAMPLE, L8, VCC, SMCLK, GND, CNTLD, CNTTC, START, IDLE, GND)];
END;

SYM GLB A4 1;
//      This is the Flip Flop to store the lower 4 bits of the bar's width
FD24 ([C0..C3]), [L0..L3], LATCH, RST);
END;

SYM GLB A5 1;
//      This is the Flip Flop to store the upper 4 bits of the bar's width
FD24 ([C4..C7]), [L4..L7], LATCH, RST);
END;

SYM GLB A6 1;
//      This is the Flip Flop to store the MSB of the bar's width
FD21 (C8, VCC, LATCH, RST);
END;
```

Figure 17. Portion of LDR file

```

SYN GLB A3 I;
// 8-bit up/down counter with Async prescaler, parallel load,
// enable, up/down, Async and Sync clear. Uses 3 GLBs
// Used as part of 9-bit counter to store the first bar's width value.
// Bit 3 is located in GLB A3.
CHUD3 ((C0..C7), (C0..C7), VCC, SMCLK, GND, CHUD3, ACTIVE, START, IDLE, GND));
END;

SYN GLB A3 I;
// This is bit 3 of the 9-bit value storing the first bar's width
CHUD1 ((8, SAMPLE, 18, VCC, SMCLK, GND, CHUD1, CHUD3, START, IDLE, GND));
END;

SYN GLB A4 I;
// This is the flip flop to store the lower 4 bits of the bar's width
FD04 ((C0..C3), (C0..C3), LATCH, RST);
END;

SYN GLB A5 I;
// This is the flip flop to store the upper 4 bits of the bar's width
FD04 ((C4..C7), (C4..C7), LATCH, RST);
END;

SYN GLB A6 I;
// This is the flip flop to store the MSB of the bar's width
FD01 ((C8, VCC, LATCH, RST);
END;

```

Lattice™ High Density PLD Solutions For High Speed RISC/CISC Systems

As the next generation Pentium™, PowerPC™ and Alpha™ processors reach new heights in speed, designers face increasingly difficult system design problems when trying to realize each processor's full speed capability. ASIC solutions are a viable option in terms of speed, but the decision to go with such a solution is influenced by another variable which is becoming ever more important - time-to-market.

Winners of the race to introduce a new product stand to reap the lion's share of profits from the product's life-cycle. Although ASICs are capable of keeping pace with the new generation of processors, they are often losers in the race for time to market. Low-density PLDs are a popular choice because of their speed and ease of programming. Another option now exists which provides the benefits of low-density PLDs while supplying higher densities and more I/Os. Lattice Semiconductor offers two new families of high speed programmable logic devices, the ispLSI and pLSI 2000 family and the ispLSI and pLSI 3000 family to address these speed and time-to-market issues. Both these families, as well as the ispLSI and pLSI 1000 family, are available as in-system reprogrammable devices which eliminate the need for sockets that often result in unreliable operation due to bent leads.

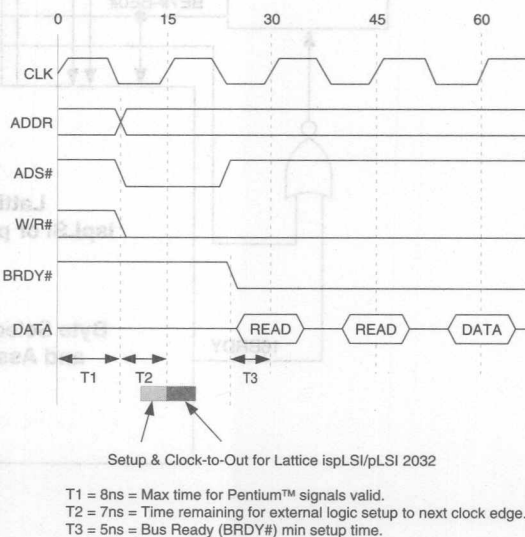
A Case For Speed

A rule of thumb for designers is that microprocessor systems typically require external logic to operate at twice the speed of the processor clock. It stands to reason that if the external logic was to consume the entire clock cycle for some computation, there would be no remaining time to satisfy any system setup requirements. This rule implies that Pentium, with its 15ns clock cycle, requires logic devices which have a speed rating of 7.5ns. If analyzed in more detail, Pentium, which has a maximum clock-edge to control-signal-valid delay of 8ns¹ (figure 1), retains 7ns of its clock cycle for external logic to perform a computation and satisfy setup requirements if a registered action is expected on the next clock edge. Logic devices with a 7.5ns Tpd rating typically have setup times in the 4ns range. Thus, such devices can realistically conform to Pentium's bus specifications. Another constraint is that external logic is to provide valid output signals in time to meet Pentium's setup require-

ment of 5ns. This implies that logic devices must have a clock-to-out time of no more than 10ns. As seen in figure 1, 7.5ns logic devices have a clock-to-out time of 4ns to 5ns which easily satisfies Pentium's setup requirements. While 7.5ns speeds are attainable from fast low-density PLDs, the requirements of today's wide buses make it desirable to have higher levels of integration with more I/O. Address decoders and bus logic are examples of circuits which demand such speed, density and I/O.

Lattice's ispLSI and pLSI 2000 family specifically targets these applications. This family has devices which are able to integrate up to 10 traditional PLDs into a single package while supplying up to 102 I/Os. These devices, while being much higher density than PLDs, suffer no speed penalty. With propagation delays of 7.5ns and clock rates of 135 MHz, the ispLSI and pLSI 2000 family operates comfortably in systems which were once the sole domain of ASICs and the fastest low density PLDs. The set-up times of the 2000 family devices are within the 7ns requirement of the Pentium bus thus allowing the generation of registered control signals such as Bus Ready (BRDY#) (figure 1).

Figure 1. Pentium™ Burst Read-Cycle With Relative Timings



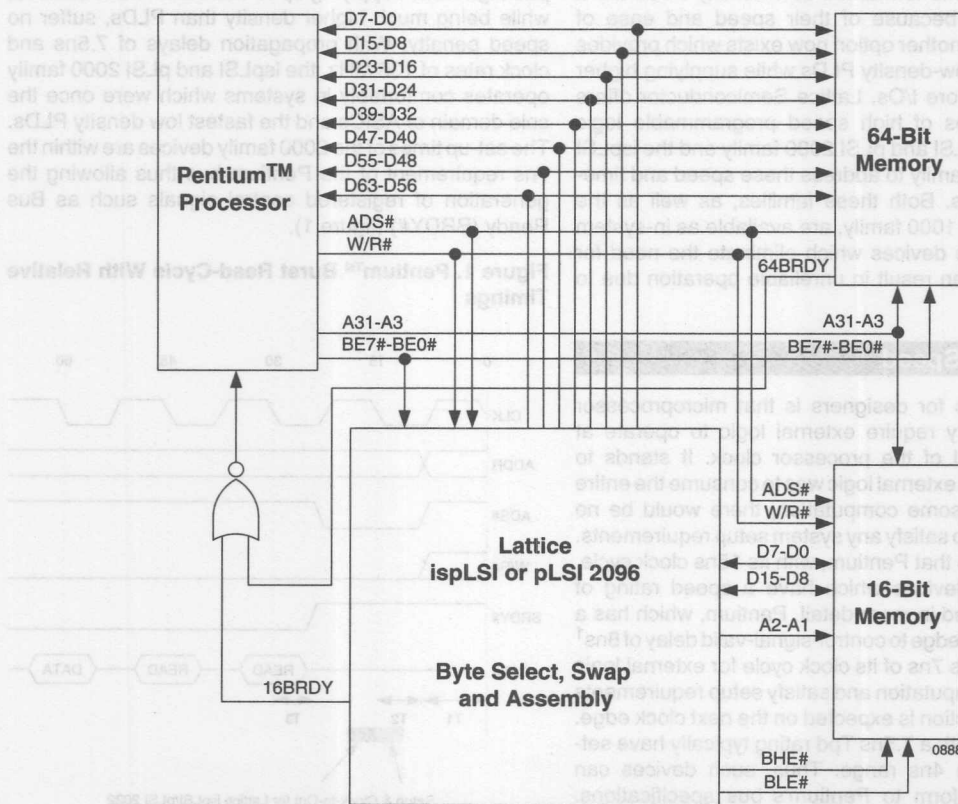
High Density PLD Solutions For High Speed RISC/CISC Systems

Address decoders are circuits which usually sit directly in the critical path of memory accesses. For this reason, T_{pd} is all important. For example, if we were to use Pentium with its 8ns Address-Valid time, a 7.5ns logic device for the address decoder, and a RAM with a 10ns Chip-Enable to Data-Out time, we would generate valid read data 25.5ns after the start of the read cycle. Pentium requires that read data be valid no later than 26.2ns (assuming no wait states). With a 7.5ns logic device, we have met Pentium's requirements with 0.7ns to spare. The ispLSI and pLSI 2032 and 2064 are designed to economically implement circuits such as this. These devices provide 32 and 64 macrocells respectively with 34 and 68 signal pins.

A Design Example

An example of a design which requires both high speed and a high degree of integration is a circuit to interface a 16-bit wide memory into a Pentium system with its 64-bit wide data bus. This 16-bit memory might possibly be a memory-mapped I/O device or a specialized RAM subsystem. The Lattice ispLSI and pLSI 2096 can be effectively used to integrate this 16-bit memory system into the Pentium's 64-bit environment while meeting all speed requirements. Figure 2 shows a block diagram of how this high-density PLD can be used to integrate the logic functions required for such an interface.

Figure 2. Interfacing A 16-bit Memory Sub-System To Pentium™

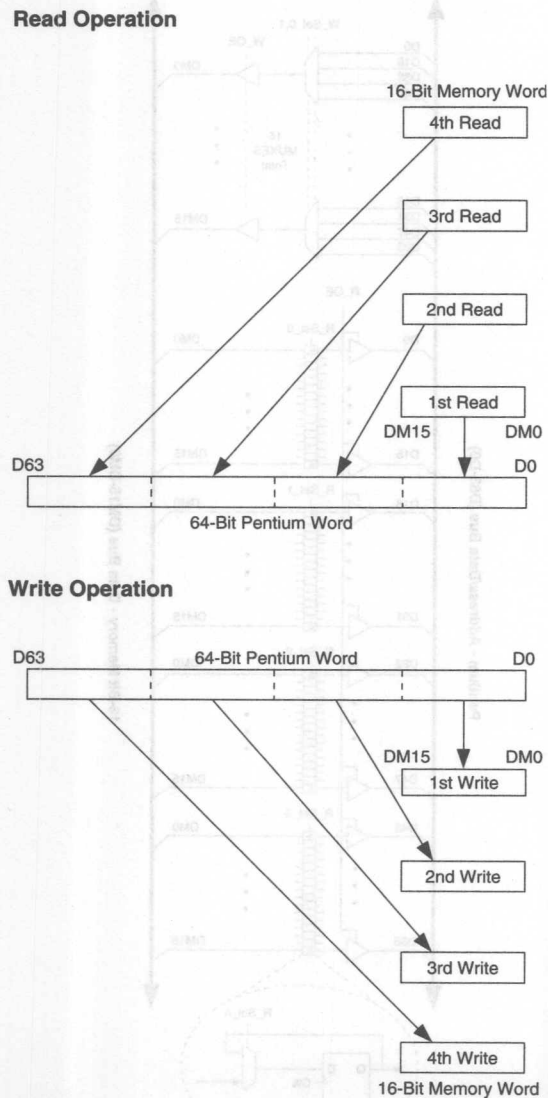


High Density PLD Solutions For High Speed RISC/CISC Systems

Intel specifies that to perform a read from a 16-bit memory, external logic is required to assemble four consecutive 16-bit reads into one 64-bit word. Writes require that the 64-bit word be broken into four 16 bit words which are consecutively written. These read and

write operations are depicted in figure 3. In addition, the Pentium decodes the three least-significant address bits into eight unary byte-select signals. These byte select signals must be re-encoded into the binary A2-A1 address bits.

Figure 3. Read And Write Operations

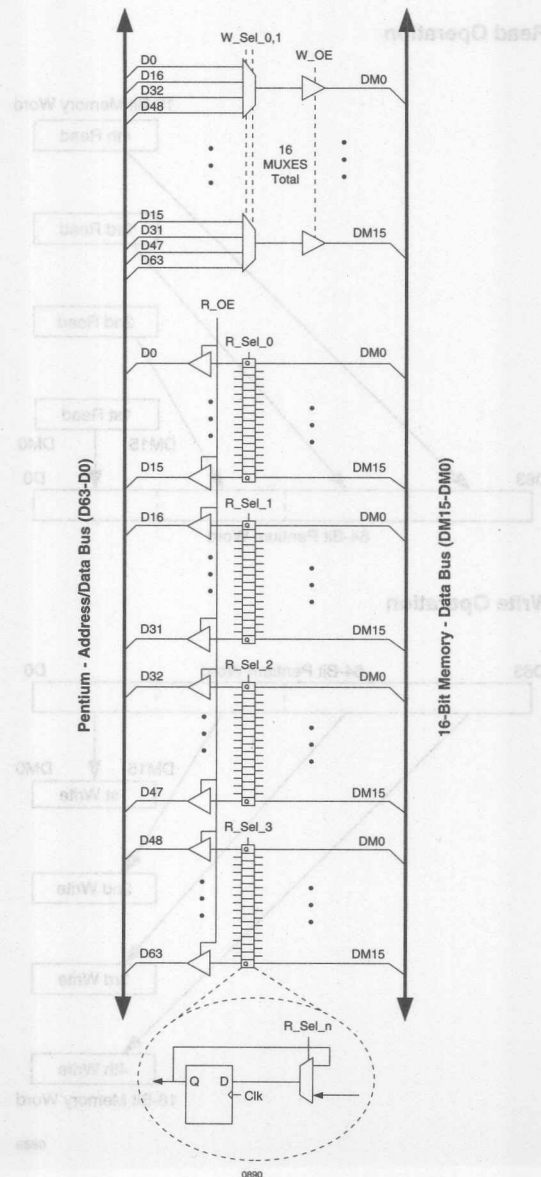


High Density PLD Solutions For High Speed RISC/CISC Systems

Within the 2096, the 16-bit write data path is created by the use of sixteen 4-to-1 multiplexers which select one of four Pentium byte pair bits. These multiplexer outputs form a 16-bit word which is written into the 16-bit memory. This circuit is shown in the upper portion of figure 4. Four

such writes occur for every one 64-bit Pentium word. The 64-bit read data path is implemented with four sequential reads of the 16-bit memory into a 64 bit wide register. The read path is shown in the lower portion of figure 4.

Figure 4. Memory Interface Data Path

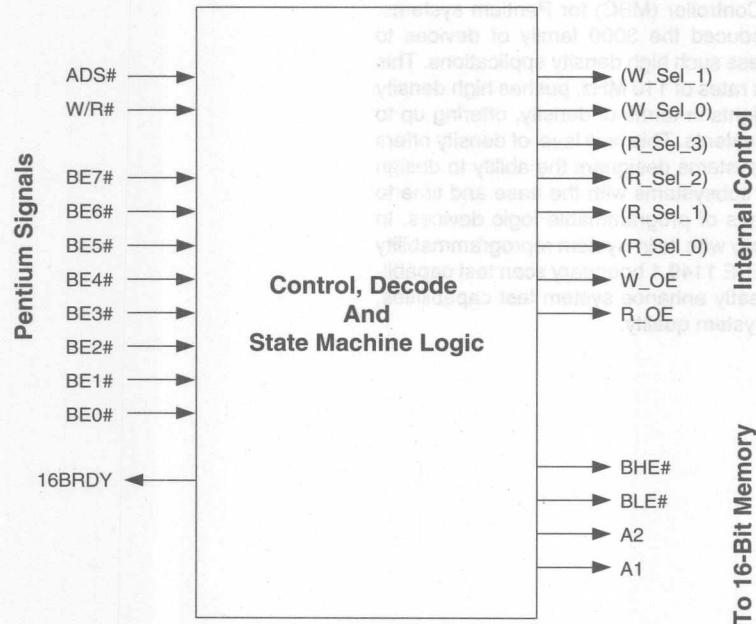


High Density PLD Solutions For High Speed RISC/CISC Systems

Control of the read and write operations is accomplished with a portion of the 2096 dedicated to control, decode and state machine functions. This control unit is shown as a block diagram in figure 5. The state machine conceptually generates the W_Sel_n and R_Sel_n control signals. These signals in actuality do not exist. Instead they are

locally decoded from state bits in the 4 to 1 MUXes and in the read register. This is done to eliminate an additional pass through the HDPLD logic and thereby improves speed. The 16BRDY signal is generated by the state machine and signals that the Pentium can advance to the next transaction.

Figure 5. Memory Interface Control Logic



Note: Signals in parenthesis are shown for logical clarity.
In practice these signals are encoded as state-bits.

0891

For full 64-bit Pentium reads and writes, the BHE# and BLE# signals are always active. A2 and A1 are sequentially incremented to provide the read and write operations as shown in figure 3. For aligned 32-bit and 16-bit reads and writes, BHE# and BLE# also remain active and the BEn bits are decoded to generate appropriate A2 and A1 bits. The sequence of address counts is reduced to 2 counts for 32-bit operations and no counts for 16-bit

operations. 8-bit operations require similar decoding but only the appropriate BHE# or BLE# signal is activated.

All these circuits can be efficiently and simply implemented in a single Lattice ispLSI and pLSI 2096 device. This interface requires 95 signal pins for which the 2096 provides 102 inputs and I/Os. 96 macrocells are available of which approximately 90 are required for this design.

High Density PLD Solutions For High Speed RISC/CISC Systems

For Higher Density Functions

Although the ispLSI and pLSI 2000 family addresses many of the needs in today's microprocessor systems, more complex subsystems may require a greater amount of logic than is available in the ispLSI and pLSI 2000 family. Such subsystems might include graphics functions, multiprocessor support, bus adapters, etc. For example, Intel does not at the time of this writing supply a Memory Bus Controller (MBC) for Pentium systems. Lattice has introduced the 3000 family of devices to specifically address such high density applications. This family, with clock rates of 110 MHz, pushes high density PLDs to new heights in terms of density, offering up to 14,000 gate equivalents. This new level of density offers microprocessor systems designers the ability to design gate-array class subsystems with the ease and time to market advantages of programmable logic devices. In addition, this family with its in-system reprogrammability and dedicated IEEE 1149.1 boundary scan test capability, is able to greatly enhance system test capabilities, thus improving system quality.

1. "Bus Functional Description", Pentium™ Processor User's Manual, Volume One. Intel Literature Number 241428. 1993.



SCSI Interface with the ispLSI 3256

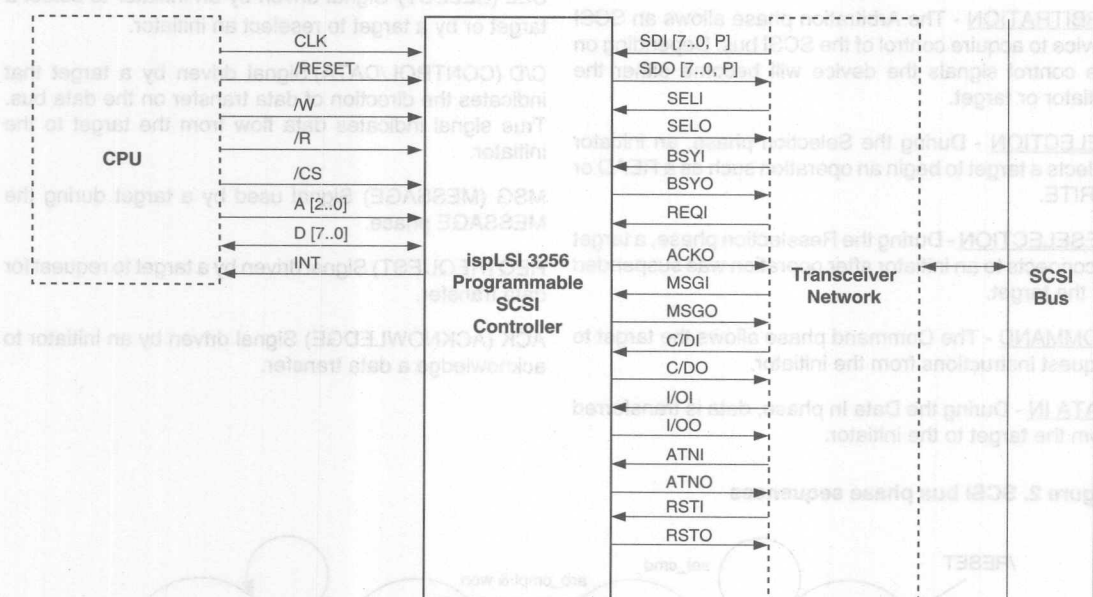
Introduction

Today's high performance computer systems require greater data storage capacity and higher throughput. The SCSI (Small computer System Interface) bus interface has risen to become the standard in peripheral communications for high-end computer systems. The versatility and flexibility of SCSI allow for higher integration without sacrificing cost and space. This applications note describes the implementation of a Programmable SCSI Controller (PSC) using Lattice's ispLSI 3256 Device. Figure 1 shows a block diagram of a PSC application.

SCSI is an intelligent bus interface that provides high performance data transfers between the host computer and peripheral devices. SCSI allows a maximum of eight devices to be attached to the bus without any additional hardware. Control of the SCSI bus is shared through arbitration using a prioritized ID assigned to each SCSI device. When two SCSI devices communicate, one acts as an initiator, and the other acts as a target. The initiator originates an operation, and the target executes the operation. The ispLSI 3256 PSC design implements a SCSI initiator.

4

Figure 1. Application Using an ispLSI 3256 Device as Programmable SCSI Controller



0879

SCSI Interface with the ispLSI 3256

SCSI Bus Phases

The communication between the devices is governed by the SCSI bus phases. Figure 2 shows a simple state flow for the different SCSI bus phases. Initial system power up and all SCSI reset conditions puts the SCSI bus in the Bus Free state. Although optional, almost all SCSI systems support and utilize the arbitration facility to prevent bus contention. Either the Selection or Reselection phases follow after winning arbitration. The Information Transfer state is really composed of the Command, Data, Status and Message phases. These specific phases determine the type of data on the bus and in what direction the data travel.

BUS FREE - The Bus Free phase indicates that the SCSI bus is available for use and that no SCSI device is actively using it. SCSI operations normally start and end with the Bus Free phase.

ARBITRATION - The Arbitration phase allows an SCSI device to acquire control of the SCSI bus. Depending on the control signals the device will become either the initiator or target.

SELECTION - During the Selection phase, an initiator selects a target to begin an operation such as a READ or WRITE.

RESELECTION - During the Reselection phase, a target reconnects to an initiator after operation was suspended by the target.

COMMAND - The Command phase allows the target to request instructions from the initiator.

DATA IN - During the Data In phase, data is transferred from the target to the initiator.

DATA OUT - During the Data Out phase, data is transferred from the initiator to the target.

STATUS - The Status phase allows the target to pass status information to the initiator.

MESSAGE IN - During the Message In phase, the target sends message(s) to the initiator.

MESSAGE OUT - During the Message Out phase, the initiator sends message(s) to the target.

SCSI Bus Signals

The SCSI bus phases are determined by the configuration of the control signals.

BSY (BUSY) Signal indicating the SCSI bus is being used by a device.

SEL (SELECT) Signal driven by an initiator to select a target or by a target to reselect an initiator.

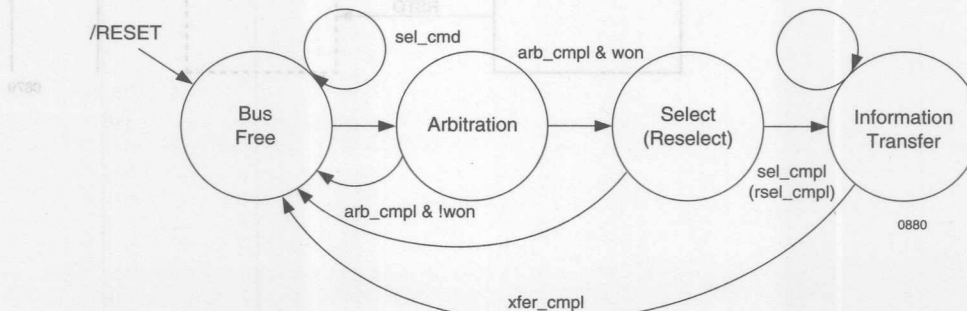
C/D (CONTROL/DATA) Signal driven by a target that indicates the direction of data transfer on the data bus. True signal indicates data flow from the target to the initiator.

MSG (MESSAGE) Signal used by a target during the MESSAGE phase.

REQ (REQUEST) Signal driven by a target to request for data transfer.

ACK (ACKNOWLEDGE) Signal driven by an initiator to acknowledge a data transfer.

Figure 2. SCSI bus phase sequences



SCSI Interface with the ispLSI 3256

ATN (ATTENTION) Signal driven by an initiator to indicate the ATTENTION condition.

RST (RESET) Signal that indicates the RESET condition.

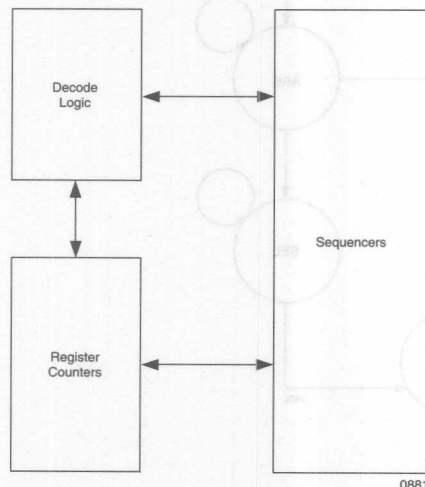
DB(7-0,P) (DATA BUS) Data bus signals that include eight data-bits and a parity-bit. Data parity is odd.

Design Description

A complex programmable logic device such as the ispLSI 3256 is an ideal solution for a Programmable SCSI Controller (PSC) device where flexibility is attained without sacrificing speed or density. The 11,000 PLD gates ispLSI 3256 with 80 MHz operating frequency and 15ns delay provides optimal performance for this application. In addition, the ispLSI 3256 provides not only in-system programmability but also reconfigurability without the need to remove the device from the PCB. The following discussion shows an ispLSI 3256 device implementing the role of an initiator with support for arbitration, selection, and reselection sequences.

Figure 3 shows the functional blocks of the ispLSI 3256 design which consists of three main modules: sequencers, decoding logic, registers and counters.

Figure 3. Block Diagram of Programmable SCSI Controller



The sequencers module consist of five state machines which process the SCSI bus data and control the flow of information. The RESEL_SM state machine handles the reselection phase sequences. The SEL_SM state machine processes all control signals and executes the selection of the target device. The ARB_SM state machine supports the arbitration phase. The DDXFER_SM state machine controls the transfer protocol between the initiator and the target. The SEQ_SM state machine is the main sequencer which oversees all other state machines.

The ispLSI 3256 architecture is ideal for building complex state machines. State transitions and conditional branching are supported by the AND-OR arrays and Product-Term Sharing Arrays (PTSA) logic. With up to 20 product-terms and hard-XOR gates, high speed complex combinatorial logic can be realized. The PSC's state machines require a large number of inputs and many product-terms to implement. With 24 inputs per GLB, the ispLSI 3256 can maintain single delay levels for high fan-in functions.

The registers of the PSC include: CMD_REG, IN_REG, OUT_REG, STAT_REG and INTR_REG. The CMD_REG (address 0) is a write-only register used to store the commands for the PSC. The IN_REG, utilizing the input registers of the ispLSI 3256, holds all the input signals from the SCSI bus. The OUT_REG stores data to be sent to the SCSI bus. The STAT_REG is a readable register giving status of the PSC and the SCSI operation.

Input Register (IN_REG)

Output Register (OUT_REG)

Command Register (CMD_REG)

SEL_CMD

Status Register (STAT_REG)

Interrupt Register (INTR_REG)

SCSI ID Register (SID_REG)

In addition to the 256 GLB registers, the ispLSI 3256 offers 128 registers/latches in the I/O cells. Besides implementing input latches (as used in the PSC design), the I/O registers/latches can also be used for signal synchronization, double registering for metastability, etc.

There are a number of counters in the PSC used to provide timing delays associated with the SCSI operations. The BSFR_DLY counter provides the necessary delay before arbitration may begin. The BSST_DLY counter provides the bus settle delay before the PSC may

SCSI Interface with the ispLSI 3256

be reselected by the target device. The ARB_DLY counter gives the arbitration delay timing.

Fast loadable counters can be easily implemented in an ispLSI 3256 device. Wide GLB inputs allow up to 24 signals including counter load inputs and Q feedbacks to drive single logic level flip-flop equations without using additional logic.

The ispLSI 3256 also provides two Global Output Enables (GOEs) which are dedicated input pins driving all of the 127 I/O cells for output or directional operations. In the PSC design, the two GOEs can be used for transferring bidirectional data to the CPU and SCSI buses without requiring internal product-terms or routing resources.

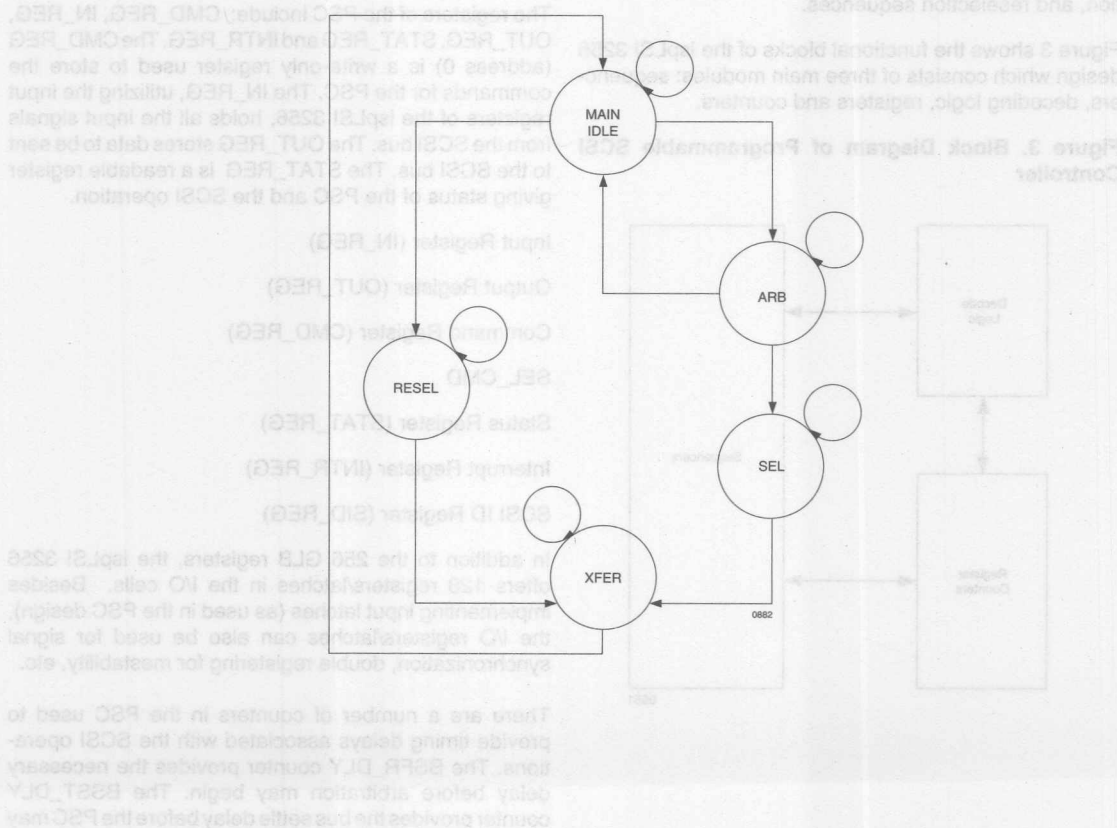
Typical SCSI Operations

A typical SCSI operation can be used to illustrate the functionality of the Programmable SCSI Controller. The first section describes sequences associated with the target selection by the PSC. The second section details the reselection of the PSC by the target device.

PSC Selects Target

INITIALIZATION - For the PSC, a typical SCSI operation starts with the SEQ_SM (main sequencer) state machine in the IDLE state and waits for a SEL command from the CPU. Figure 4 shows a state machine diagram for SEQ_SM. Once the SEL command is received, the SEQ_SM goes into the arbitrate state and remain there until arbitration is complete. Listing 1 details the ABEL implementation of the main sequencer.

Figure 4. Main Sequencer State Machine Diagram (SEQ_SM)



SCSI Interface with the ispLSI 3256

Listing 1. Before arbitration can begin, the arbitration process must be entered.

```

STATE DIAGRAM main_sm
STATE main_idle:
  IF sel_cmd==1 THEN arb_st
  " Receive Select command "
  WITH arb_phs.d = 1
  ELSE IF (resel_phs) THEN resel_st
  " Detect Reselection Phase "
  ELSE main_idle;

STATE arb_st:
  " Arbitration Phase "
  IF (arb_cmpl & won) THEN sel_st
  " Won Arbitration "
  WITH sel_phs.d = 1
  " Goto Selection Phase "
  ELSE IF (arb_cmpl & !won) THEN
    main_idle " Lost Arbitration "
  WITH intr.d = 1;
  " Interrupt CPU "

STATE sel_st:
  " Selection Phase "
  IF (sel_cmpl) THEN xfer_st

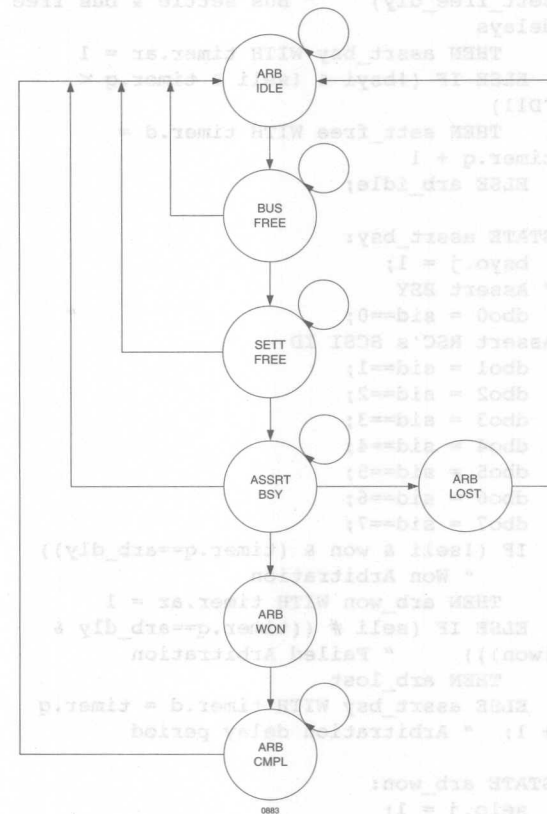
  WITH xfer_phs.d = 1
  ELSE sel_st;

STATE resel_st:
  " Reselection Phase "
  IF (resel_cmpl) THEN xfer_st
  WITH xfer_phs.d = 1
  ELSE resel_st;

STATE xfer_st:
  " Data Transfer Phase "
  IF (xfer_cmpl) THEN main_idle
  ELSE xfer_st;

END
  
```

Figure 5. Arbitration State Machine Diagram (ARB_SM)



4

Listing 2.

```

STATE DIAGRAM arb_sm
STATE arb_idle:
  IF (arb_phs) THEN bus_free
  " Detect start of Arbitration "
  ELSE arb_idle;

STATE bus_free:
  IF (!bsyi & !seli) THEN sett_free
  " Detect Bus Free state "
  WITH timer.ar = 1
  ELSE IF (arb_phs) THEN bus_free
  ELSE arb_idle;
  
```


SCSI Interface with the ispLSI 3256

```

STATE sett_free:
  IF (!bsyi & !seli & timer.q ==
sett_free_dly) " Bus settle & bus free
delays
  THEN assrt_bsy WITH timer.ar = 1
  ELSE IF (!bsyi & !seli & timer.q <
^D11)
  THEN sett_free WITH timer.d =
timer.q + 1
  ELSE arb_idle;

STATE assrt_bsy:
  bsyo.j = 1;
  " Assert BSY
  dbo0 = sid==0;
  Assert RSC's SCSI ID
  dbo1 = sid==1;
  dbo2 = sid==2;
  dbo3 = sid==3;
  dbo4 = sid==4;
  dbo5 = sid==5;
  dbo6 = sid==6;
  dbo7 = sid==7;
  IF (!seli & won & (timer.q==arb_dly))
  " Won Arbitration
  THEN arb_won WITH timer.ar = 1
  ELSE IF (seli # ((timer.q==arb_dly &
!won))) " Failed Arbitration
  THEN arb_lost
  ELSE assrt_bsy WITH timer.d = timer.q
+ 1; " Arbitration delay period

STATE arb_won:
  selo.j = 1;
  " Assert SEL
  GOTO arb_cmpl;

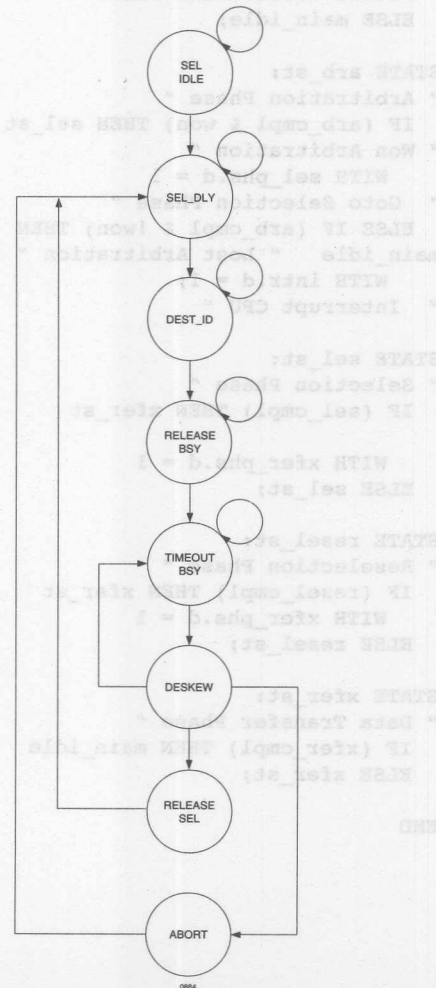
STATE arb_cmpl:
  IF (timer.q==sett_clr_dly)
  " Wait for bus settle & clear
  THEN arb_idle WITH arb_cmplt = 1
  " End of Arbitration
  ELSE arb_cmpl WITH timer.d = timer.q +
1;

STATE arb_lost:
  arb_cmplt = 1;
  " End of Arbitration
  bsyo.k = 1;
  " Negate BSY
  arb_fail_int = 1;
  " Set Interrupt
  GOTO arb_idle;
END

```

BUS FREE PHASE - Before arbitration can begin, the PSC must detect the Bus Free phase. The ARB_SM state machine must read the BSY and SEL signals false for a bus free delay until the arbitration phase is entered. Figure 5 shows a state machine diagram for ARB_SM and Listing 2 details the ABEL implementation of the arbitration process.

Figure 6. Selection State Machine Diagram (SEL_SM)



Listing 3.

STATE DIAGRAM selection

```

STATE sel_idle_st:
  IF (sel_phs) THEN sel_dly_st
  ELSE sel_idle_st;

STATE sel_dly_st:
  IF (sel_dly_cnt == 12) THEN dest_id_st
  ELSE sel_dly_st
  WITH sel_dly_cnt.D = sel_dly_cnt.Q +
1;

STATE dest_id_st:
  dbo0 = (dest_id==0);
  dbo1 = (dest_id==1);
  dbo2 = (dest_id==2);
  dbo3 = (dest_id==3);
  dbo4 = (dest_id==4);
  dbo5 = (dest_id==5);
  dbo6 = (dest_id==6);
  dbo7 = (dest_id==7);
  IF (id_dly_cnt==4 & atn_cmd) THEN
  release_bsy_st WITH atno=1
  ELSE IF (id_dly_cnt==4 & !atn_cmd)
  THEN release_bsy_st
  ELSE dest_id_st WITH
  id_dly_cnt.D=id_dly_cnt.Q+1;

STATE release_bsy_st:
  bsyo.K = 1;
  IF (bsy_dly_cnt==4) THEN

```

```

timeout_bsy_st
  ELSE release_bsy_st WITH bsy_dly_cnt.D
  = bsy_dly_cnt.Q + 1;

STATE timeout_bsy_st:
  IF (bsyi & resel_cmd) THEN deskew_st
  ELSE if (!bsyi & timeout_cnt==timeout)
  THEN abort_st
  ELSE timeout_bys_st WITH timeout.D =
  timeout.Q + 1;

STATE deskew_st:
  GOTO deskew1_st;

STATE deskew1_st:
  GOTO deskew2_st;

STATE deskew2_st:
  GOTO release_sel_st;

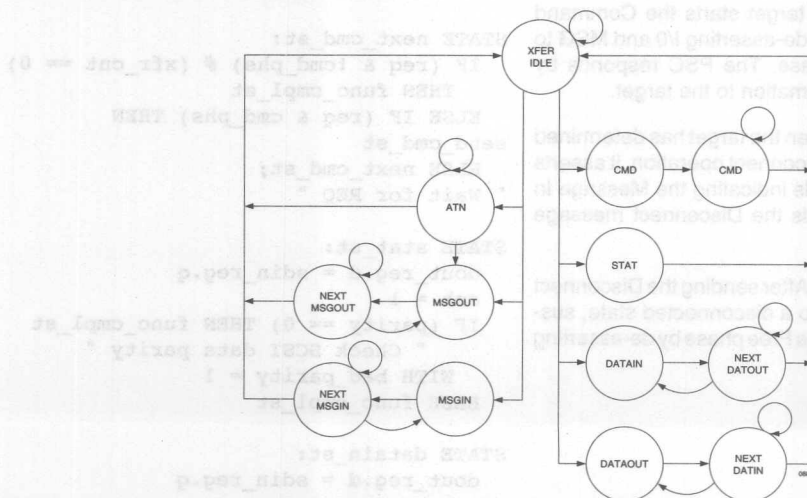
STATE release_sel_st:
  selo.K = 1;
  sel_cmplt = 1;
  GOTO sel_idle_st;

STATE abort_st:
  selo.K = 1;
  disc_int.D = 1;
  cmd_reg.re = 1;
  GOTO sel_idle_st;

END

```

Figure 7. Data Transfer State Machine Diagram (XFER_SM)



SCSI Interface with the ispLSI 3256

ARBITRATION PHASE - In the arbitration phase, one or more devices will try to gain control of the SCSI bus. In state machine ARB_SM shown in figure 5 and Listing 2, the PSC asserts BSY and drives its SCSI ID bit onto the 8-bit data bus. At the same time, the PSC reads the data bus to determine whether a device with higher priority wants control of the bus. The highest priority device wins control of the bus and continues to assert the BSY and SEL signals. All other devices participating in the arbitration must release BSY and their SCSI ID bit when SEL become active.

SELECTION PHASE - After winning the arbitration, the PSC (acting as initiator) asserts both SEL and BSY signals. Figure 6 shows the Selection state diagram and Listing 3 details the ABEL implementation. In state machine SEL_SM, to select a target, the PSC releases the BSY signal, drives the target's SCSI ID bit and its own ID bit active on the data bus, and de-asserts the I/O signal. The PSC will continue to drive SEL until the target asserts BSY.

ATTENTION CONDITION - The PSC may assert the ATN signal during the Selection phase and while the SEL signal is still asserted, thus indicating that it wants the target to go to the Message Out phase immediately after the Selection phase.

MESSAGE OUT PHASE - Figure 7 and Listing 4 show the Data Transfer State Machine. During the Message Out phase, the target asserts CD and MSG and de-asserts I/O. The PSC sends the Identify message to indicate which logical unit of the target is to be selected and that the PSC supports the Disconnect/Reselect operation.

COMMAND PHASE - The target starts the Command phase by asserting CD and de-asserting I/O and MSG to indicate the Command phase. The PSC responds by sending the command information to the target.

MESSAGE IN PHASE - When the target has determined that it needs to perform a disconnect operation, it asserts the CD, I/O and MSG signals indicating the Message In phase. The PSC then reads the Disconnect message from the target.

DISCONNECTED STATE - After sending the Disconnect message, the target goes to a disconnected state, suspending operations in the Bus Free phase by de-asserting all control signals.

Listing 4

```
STATE DIAGRAM xfer_sm
STATE xfer_idle:
  IF (atno == 1) THEN xfer_idle
  WITH atno.k = 1
  " Clear ATN "
  ELSE IF (atn_cmd) THEN atn_st
  " Attention requested "
  WITH atno.j = 1
  " Assert ATN "
  ELSE IF (reqi & cmd_phs) THEN cmd_st
  ELSE IF (reqi & stat_phs) THEN stat_st
  ELSE IF (reqi & datin_phs) THEN
    datin_st
  ELSE IF (reqi & datout_phs) THEN
    datout_st
  ELSE IF (reqi & msgout_phs) THEN
    msgout_st
  ELSE IF (reqi & msgin_phs) THEN
    msgin_st
  ELSE xfer_idle;
STATE atn_st:
  IF (req & msgout_phs) THEN msgout_st
  ELSE IF (req & lmsgout_phs)
    THEN func_cmpl_st
  ELSE atn_st;
STATE cmd_st:
  sdout_reg.d = din_reg.q
  xfr_cnt.d = xfr_cnt.q - 1
  ack = 1
  GOTO next_cmd_st;
STATE next_cmd_st:
  IF (req & !cmd_phs) # (xfr_cnt == 0)
    THEN func_cmpl_st
  ELSE IF (req & cmd_phs) THEN
    send_cmd_st
  ELSE next_cmd_st;
  " Wait for REQ "
STATE stat_st:
  dout_reg.d = sdin_reg.q
  ack = 1
  IF (parity == 0) THEN func_cmpl_st
  " Check SCSI data parity "
  WITH bad_parity = 1
  ELSE func_cmpl_st
STATE datain_st:
  dout_reg.d = sdin_reg.q
```

```

ack = 1
IF (parity == 0) THEN func_cmpl_st
  WITH bad_parity = 1
ELSE next_datin_st;

STATE next_datin_st:
  IF (req & !datain_phs) # (xfr_cnt == 0)
    THEN func_cmpl_st
  ELSE IF (req & datain_phs) THEN
    datain_st
  ELSE next_datin_st;

STATE dataout_st:
  sdout_reg.d = din_reg.q
  xfr_cnt.d = xfr_cnt.q - 1
  ack = 1
  GOTO next_datout_st;

STATE next_datout_st:
  IF (req & !dataout_phs) # (xfr_cnt == 0)
    THEN func_cmpl_st
  ELSE IF (req & dataout_phs) THEN
    dataout_st
  ELSE next_datout_st;

STATE msgout_st:
  sdout_reg.d = din_reg.q
  xfr_cnt.d = xfr_cnt.q - 1
  ack = 1
  GOTO next_msgout;

STATE next_msgout:
  IF (req & !msgout_phs) # (xfr_cnt == 0)
    THEN func_cmpl_st
  ELSE IF (req & msgout_phs) THEN
    msgout_st
  ELSE next_msgout;

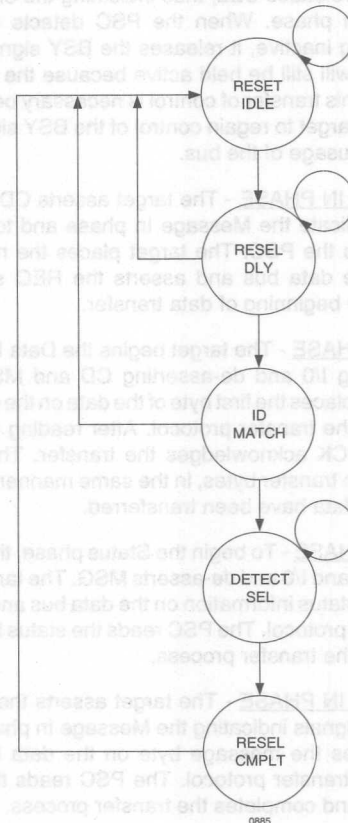
STATE msgin_st:
  dout_reg.d = sdin_reg.q
  xfr_cnt.d = xfr_cnt.q - 1
  ack = 1
  IF (parity == 0) THEN func_cmpl_st
    WITH bad_parity = 1
  ELSE next_msgin;

STATE next_msgin:
  IF (req & !msgin_phs) # (xfr_cnt == 0)
    THEN func_cmpl_st
  ELSE IF (req & msgin_phs) THEN msgin_st
  ELSE next_msgin;

END

```

Figure 8. Reselection State Machine Diagram (RESEL_SM)



Target Reselects PSC

The target remains in the disconnected state until it is ready to continue with the next SCSI operation. The PSC is also in the disconnected state until the target reselects it.

ARBITRATION PHASE - Before reselecting the PSC, the target goes through the arbitration process to acquire control of the SCSI bus. The target asserts BSY and its SCSI ID bit.

RESELECTION PHASE - The target drives its SCSI ID, and the PSC's ID on to the data bus, and then asserts SEL and I/O and de-asserts BSY. The PSC reads its SCSI ID and the control signals to determine that it has been reselected by the target. Figure 8 and Listing 5 show the Reselection State Machine.

SCSI Interface with the ispLSI 3256

When reselected, the PSC responds to the target by asserting BSY. The target then drives the BSY signal active and releases SEL, thus indicating the end of the Reselection phase. When the PSC detects the SEL signal going inactive, it releases the BSY signal. However, BSY will still be held active because the target is driving it. This transfer of control is necessary because it allows the target to regain control of the BSY signal and control the usage of the bus.

MESSAGE IN PHASE - The target asserts CD, I/O and MSG to indicate the Message In phase and to send a message to the PSC. The target places the message byte on the data bus and asserts the REQ signal to indicate the beginning of data transfer.

DATA IN PHASE - The target begins the Data In phase by asserting I/O and de-asserting CD and MSG. The target then places the first byte of the data on the data bus and starts the transfer protocol. After reading the data byte, the PCK acknowledges the transfer. The target continues to transfer bytes, in the same manner, until all requested data have been transferred.

STATUS PHASE - To begin the Status phase, the target asserts CD and I/O and de-asserts MSG. The target then places the status information on the data bus and begins the transfer protocol. The PSC reads the status byte and completes the transfer process.

MESSAGE IN PHASE - The target asserts the CD, I/O and MSG signals indicating the Message In phase. The target places the message byte on the data bus and begins the transfer protocol. The PSC reads the message byte and completes the transfer process.

BUS FREE PHASE - After sending the "Command Complete" message, the target releases control of the SCSI bus by de-asserting all control signals. After the PSC and target physically and logically disconnect from the bus, the Bus Free phase begins.

Listing 5.

```
STATE DIAGRAM reselection

STATE resel_idle_st:
    IF (resel_phs) THEN resel_dly_st;
    ELSE resel_idle_st;

STATE resel_dly_st:
    IF (timer == bsst_dly & resel_phs) THEN
        id_match_st
    ELSE IF (!resel_phs) THEN resel_idle_st
    ELSE resel_dly_st
        WITH timer.d = timer.q + 1;

STATE id_match_st:
    IF (tar_id_match & psc_id_match & parity)
        THEN detect_sel_st WITH bsyo.j = 1
    ELSE resel_idle_st;

STATE detect_sel_st:
    IF (!seli) THEN resel_cmp_st WITH
        bsyo.k = 1
    ELSE detect_sel_st;

STATE resel_cmp_st:
    resel_cmpl = 1;
    GOTO resel_idle_st;

END
```

Conclusion

Lattice's ispLSI 3256 is the ideal solution for implementing a Programmable SCSI Controller. The ispLSI 3256's input registers allows asynchronous signals to be synchronized to the PSC's system clock. The in-system programmability and reconfigurability of the ispLSI 3256 enables different SCSI configurations to be implemented or upgraded without the need to remove the device from the board.

Introduction

The Peripheral Component Interconnect (PCI) Local bus was designed as a high bandwidth bus that provides a data path between the CPU and multiple high performance peripherals. Proposed as a total system solution, PCI provides interconnects to networks, disk drives, video and other high speed peripherals. Processor independence allows the PCI bus to be optimized for I/O functions and enables concurrent operation of the local bus with the processor/memory subsystem. A 32 bit synchronous bus that provides data throughput of 132 Mbytes/sec, the PCI bus is expandable up to a 64 bit data path which doubles the throughput. On account of its futuristic processor independent orientation, PCI allows manufacturers to significantly trim development costs by not having to completely redesign every product cycle.

This ties in elegantly with the Lattice ispLSI (in-system programmable Large Scale Integration) family, designed to implement high integration functions, such as controllers, while delivering superior performance and the flexibility of In-System Programmability (ISP). The basic PCI compliant Master/Target state machines can be implemented in the ispLSI device, while the remaining glue logic can be modeled around a given peripheral/processor. The options become enormous, when one has the ability to change the functionality of devices

already soldered on the board. ISP continues to emerge as the design methodology of choice by providing reconfigurable systems with diagnostic capabilities, field upgradeability and simplification of manufacturing flow.

PCI flexibility brings with it new design challenges for the system designer. This application note presents a Master/Target-PCI interface design implemented in an ispLSI device. The attached source code contains the basic PCI compliant state machines and is intended to be used as a guideline on which a PCI bridge design for a specific interface can be based. The benefits of ispLSI as applied to the PCI bus, and AC/DC and timing specifications are reviewed.

PCI/Lattice ispLSI Interface

The following section presents the PCI interface based on the PCI Local Bus Specifications, Revision 2.0. A concise overview of the PCI bus and ispLSI architecture and the relevant electrical and timing characteristics are discussed. The Lattice 1994 Data Book and the PCI Specification should be consulted to obtain more detailed information.

PCI Overview

The PCI bus is a non-proprietary local bus solution, providing increased performance for Graphical User In-

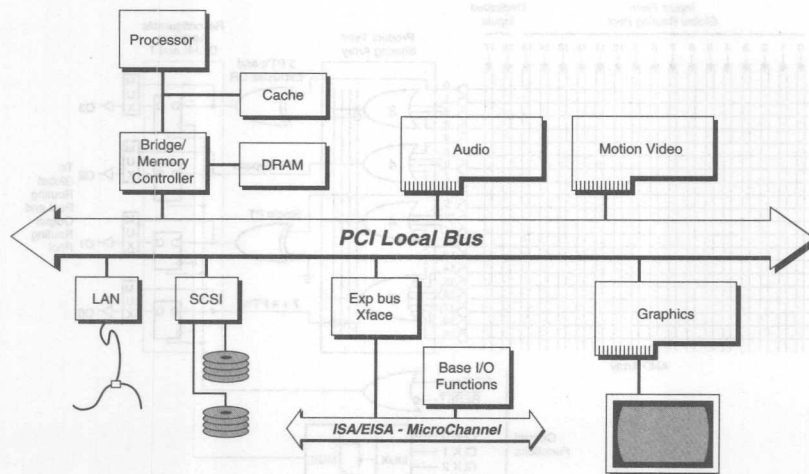


Figure 1. PCI System Block Diagram

Portions of this document were reprinted with the permission of the PCI Special Interest Group. Copyright 1992, 1993 PCI Special Interest Group.

PCI Bus Implementation

interfaces and other high bandwidth functions such as SCSI, full motion video, LANs etc.. The PCI component and the add-in card interface is processor independent, enabling an efficient transition to future processor generations and use with multiple processor architectures. Processor independence allows the bus to be optimized for I/O functions, enabling concurrent operation of the bus with the processor/memory subsystem. Figure 1 shows a typical PCI system.

The processor/memory subsystem is connected to PCI through a bridge, which provides a low latency path for

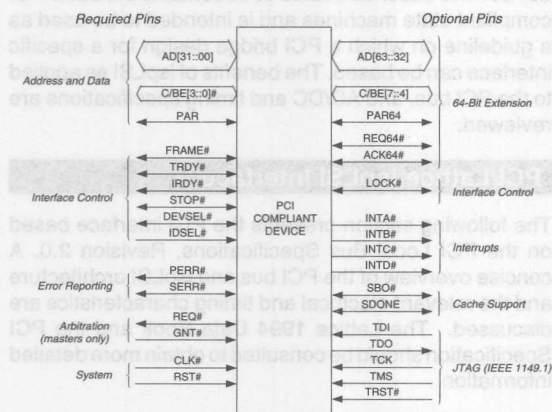


Figure 2. PCI Pin List

the agent to directly access the PCI devices mapped onto the processor address space. The PCI specifications defines both a Master and Target bridge implementation. Both can be implemented in one device, however each has to have an independent controller state machine. Figure 2 shows the pins required on a PCI controller in order to handle addressing, arbitration, interface control and other system functions. A minimum of 47 pins are needed for a Target only device and 49 pins for a Master.

The PCI interface consists of two different types of buses and control signals which govern the timing of data transfer on the address/data bus by the insertion of wait states. The larger of the two buses is the multiplexed Address/Data (AD) bus. The transfer of data onto the AD bus is not required to be the full width of the bus. The width of the data transfer is indicated by control information present at the time of the bus transaction. The second bus is the Command/Byte Enable (C/BE) bus. The C/BE bus contains information about the activity that is to occur, i.e. read/write and memory or I/O access, during the address phase of the bus transaction, and contains the byte enables during the data phase of the bus transaction. Byte lane swapping is not allowed on the PCI bus since all devices must connect to 32 address/data bits. Furthermore, automatic bus sizing is not supported and the byte enables determine which bytes carry meaningful data. The PCI bus interface requires that every active member connected to the PCI bus be synchronized to a system clock. This allows information to be transferred between the active agents with wait

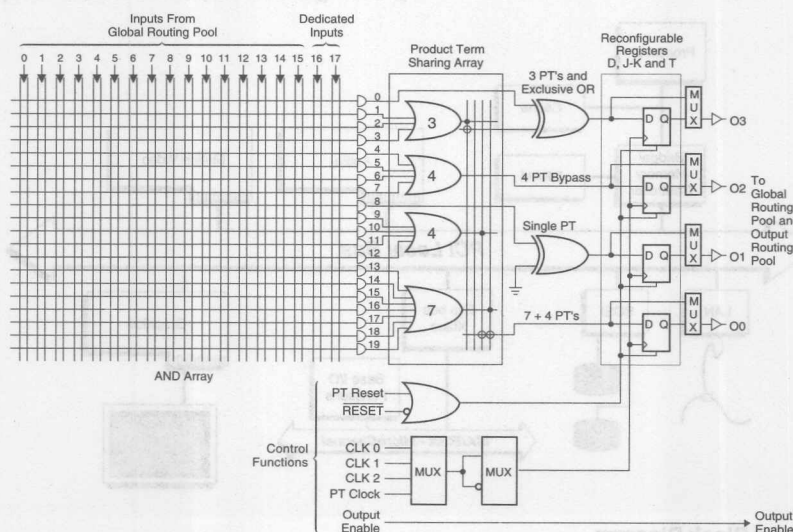


Figure 3. Mixed Mode Generic Logic Block

states, inserted by Master or Target, to match the timing requirements of either party that is involved in bus activity. The wait states are inserted through the use of the signals IRDY and TRDY. The signal FRAME indicates that a Master is currently active on the bus and that all other bus Masters are not to become active on the bus until the current activity is completed.

Lattice ispLSI Architectural Overview

Lattice's ispLSI HDPLDs are ideally suited to high speed controller, state machine intensive applications. This section provides a broad overview of the architecture. Relevant features will be discussed in further detail as they relate to this application note. In addition to In-System Reprogrammability, characteristics such as wide input gating (18 input/20 product terms per register), hardware XOR gates on each register, low skew (less than 2 ns), input clamping capability and high speed make the ispLSI device ideal for complex state machine implementation. The ispLSI devices contain programmable logic, registers, I/O pins, multiple clocks, a Global Routing Pool and Output Routing Pool. The basic unit of logic is the Generic Logic Block (GLB). Figure 3 shows a simplified logic diagram of the ispLSI GLB.

The Lattice ispLSI devices are programmable, in circuit, on a powered board. This simplifies the design flow by eliminating the time consuming simulation process. The design can be tested in the final system by downloading the JEDEC file directly into the part. This is especially useful in surface mount environments where the parts cannot be removed from the board for programming. Test points are brought out to unused I/O pins during the debug cycle, and eliminated for standard operation. A

designer can complete the design in steps by creating smaller modules of the design, testing them as stand alone circuits, and then combining them once they are all working correctly. In addition to being a design tool, In-System Programming also offers production advantages. Field service upgrades can be performed by simply reprogramming the boards, and options added by programming them into the logic. If several boards are similar in function, but have different logic, a single printed circuit board can be designed, and the specific function programmed into the logic just before the board is shipped. This reduces both production and inventory costs.

The only requirements of the system are that it must have a stable 5 volt power supply, and a connection point for the ispDOWNLOAD Cable. The standard interface used on the ispLSI prototype boards is a common 8-pin telephone connector. This connector is selected because it is small, reliable and inexpensive. Five pins on the ispLSI 1032 device are dedicated to programming when the part is used in the ISP mode. They are:

ispEN	In-System Programming Enable
MODE	ISP Mode Control
SCLK	Shift Clock
SDI	Serial Data In
SDO	Serial Data Out

The algorithm which is used to program the part is straightforward. The MODE, SCLK and SDI pins are used to control a state machine internal to the ispLSI device. The device is controlled by serially shifting in a series of commands and data streams. The state diagram for that operation is shown in Figure 4.

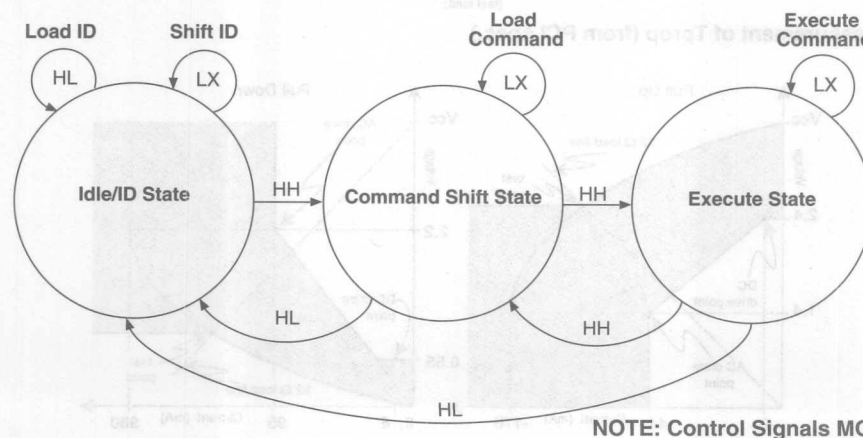


Figure 4. ISP Programming State Machine

PCI Bus Implementation

PCI Electrical Specifications

The PCI specification provides for both 5V and 3.3V signaling environments, but all components in a PCI design must use the same signaling environment. The PCI bus is a CMOS bus, i.e., steady state currents are minimal (after transients have died out), with most of the current spent on pull-up resistors. PCI is based on reflective wave signaling, rather than incident wave, which implies that the bus drivers have to switch the bus halfway to the required high or low voltage. The fact that the bus is unterminated, causes the reflected wave at the unterminated end of the transmission line to add to the incident wave to achieve the required voltage level. (See figure 5). The bus driver is actually in the middle of its switching range during this propagation time, which lasts up to 10ns, or one third the bus cycle frequency of 33MHz. The PCI bus drivers are specified in terms of the AC switching characteristics or V/I curves. Figure 6 shows the V/I curves of the PCI bus under a 5V signaling environment.

The PCI specification dictates that pins used for extended data path (64 bit) such as high order AD lines, C/BE lines and PAR64 (64-bit extension parity-- see figure

2) have pullups in order to prevent oscillation or high power drain through the input buffer. Some signals have to be pulled up in order to have stable values when no agent is driving the bus. In addition, the inputs are required to be clamped to ground. According to the PCI Local Bus specification, clamps to 5V are optional, but may be needed to protect 3.3V devices. When using dual power rails, parasitic diodes exist from one supply to another. These diode paths can become forward biased, if one of the power rails goes out of spec. for an instant. The diode clamps to the power rail and to the output devices must be able to withstand short circuit current until the drivers can be tristated.

It should be noted that PCI compliant devices that directly drive the bus have extremely high output drive capability (greater than 48mA). This high drive is required to overcome incident wave effects that may occur within the design and not so much from a DC drive perspective. Hence, the ispLSI devices may be used in conjunction with external buffers (GAL16VP8 or 20VP8) or with series termination applied. In many cases, the loading conditions are such that no external buffering or termination is needed. This must be determined by the system designer.

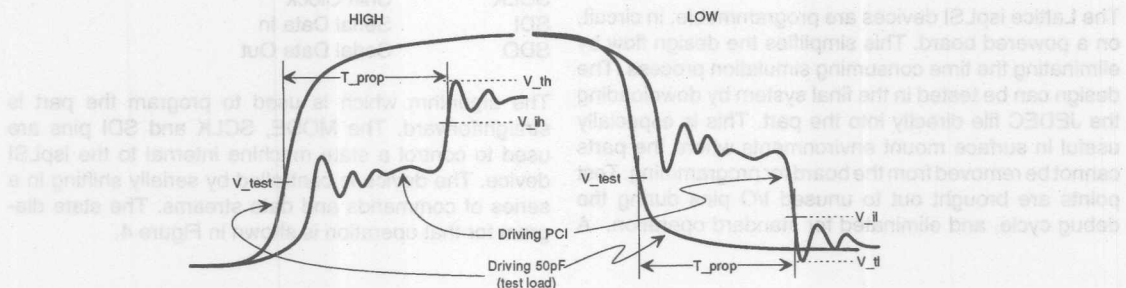


Figure 5. Measurement of Tprop (from PCI spec.)

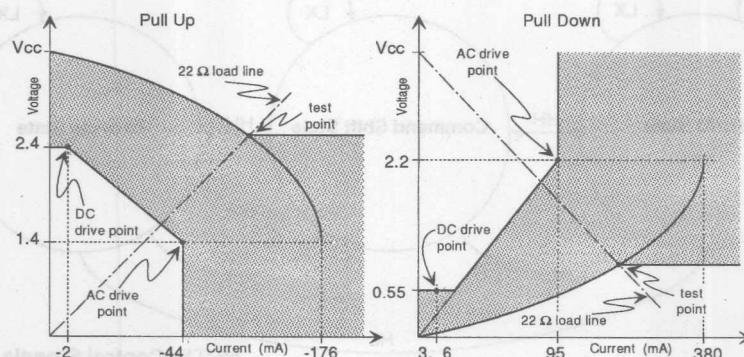


Figure 6. V/I curve for 5V signaling (from PCI spec.)

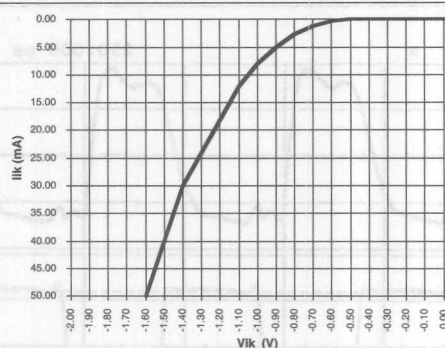


Figure 7a. ispLSI Input Clamp Characteristics

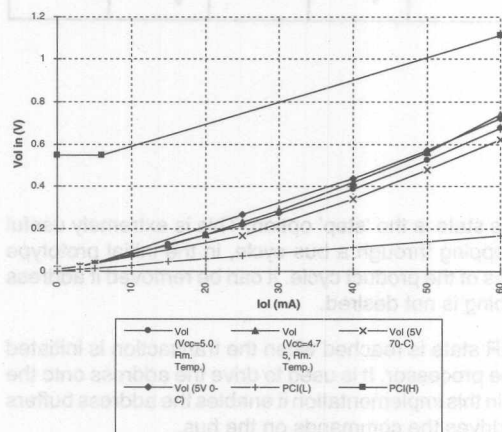


Figure 7b. ispLSI Vol vs. Iol

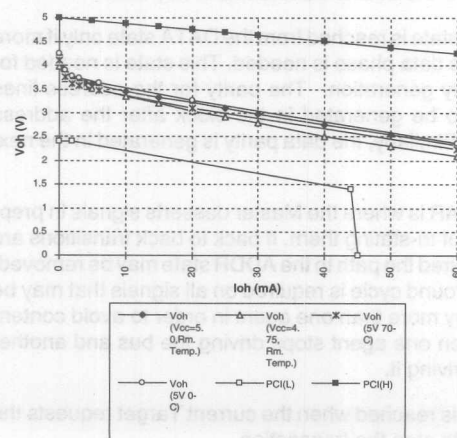


Figure 7c. ispLSI Voh vs. Ioh

Lattice ispLSI Electrical Specifications

The Lattice ispLSI family has programmable pull-up resistors that may be used instead of the external resistors, saving real estate. The ispLSI devices have an input clamp that turns on at approximately -1.7v, -18mA (see figure 7). These clamps exist on each of the dedicated inputs and I/Os. In addition, the ispLSI devices are capable of operating under conditions of "excessive" overshoot or undershoot. Figure 8 depicts the results when a 16 volt peak-to-peak pulse is injected into the input or I/O pin.

Finally, with respect to input capacitance, the PCI specification stipulates that the input capacitance should not exceed 10 pF for an input pin and 12 pF for the clock and I/O pin. The ispLSI devices have input capacitance of 8 pF on input pins and 10 pF on I/O and clock pins.

PCI Timing Requirements

The PCI specification provides strict timing requirements in terms of setup time (7ns minimum). The Lattice ispLSI 1032-80 device has a minimum set up time of 7ns on the inputs.

Please refer to the PCI specifications and the Lattice Data Book for detailed specifications of the PCI bus and Lattice ispLSI devices.

Controller Logic Implementation

This section describes the implementation of the Master and Target state machines. Simulation waveforms are provided for the read cycle in Appendix A. The equations are for illustrative purposes only, and may have to be modified to support the actual design requirements. Lattice Semiconductor Corp. is not responsible for conflicts between the design and the specification. The PCI protocol has priority if any conflict arises in the equations.

Master State machine

The PCI Master performs the following functions:

1. Data reads and writes on the PCI bus along with address stepping
2. Initiate a time-out if cycle is not decoded by any Target (no subtractive decoding)
3. Initiate a PCI bus latency time-out
4. Responds to the system reset
5. Generate parity error
6. Can address memory or I/O space
7. PCI bus locked cycles

The Master state machine supports several options as specified in the PCI protocol. The bus interface consists

PCI Bus Implementation

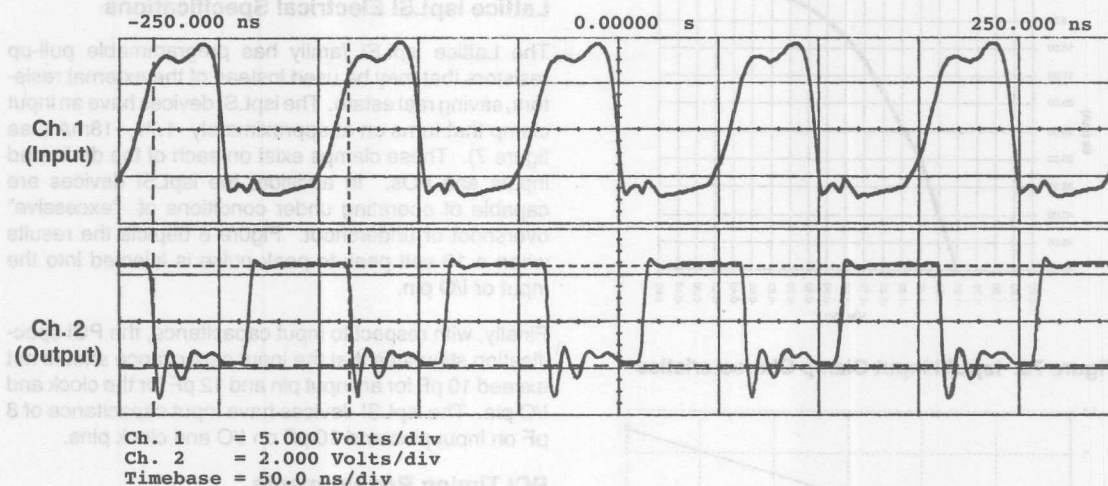


Figure 8. isPLSI Overshoot/Undershoot Characteristics

of two parts. First, the Master Sequencer state machine which actually performs the bus operation. The second part, the backend (processor), initiates the transaction and provides the address, data, command, byte enables and the length of the transfer. It is responsible for the address if the transaction is retried. The backend can request a locked transfer or terminate a transfer. Each state of the sequencer machine will be discussed, with viable options. There are seven valid states of the sequencer machine:

IDLE is when the Master waits for a request from the backend to do a bus operation. The only possible option

in this state is the 'step' option. This is extremely useful in stepping through a bus cycle, in the initial prototype stages of the product cycle. It can be removed if address stepping is not desired.

ADDR state is reached when the transaction is initiated by the processor. It is used to drive the address onto the bus, in this implementation it enables the address buffers and drives the commands on the bus.

DATA state is reached unconditionally from the ADDR state and the data is transferred in this state.

DATA1 state is reached from the DATA state only if more than one data phase is needed. This state is needed for the parity generation. The parity for the address lines needs to be generated in the clock after the address phase. Similarly, the data parity is generated in the next clock.

TURN_AR is where the Master asserts signals in preparation of tri-stating them. If back to back transitions are not required the path to the ADDR state may be removed. A turnaround cycle is required on all signals that may be driven by more than one agent in order to avoid contention when one agent stops driving the bus and another starts driving it.

S_TAR is reached when the current Target requests the Master to stop the transaction.

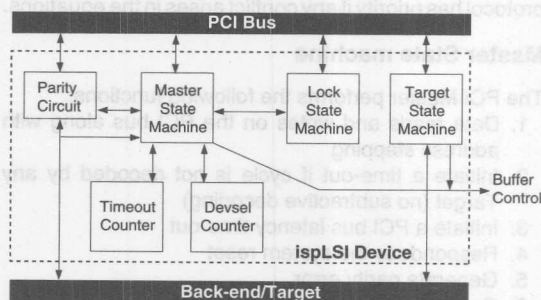


Figure 9. Controller Block Diagram

DR_BUS is used when the PCI bus has been granted to the current Master and the Master either is not prepared to start a transaction (for address stepping) or has none pending.

The following is the state diagram for the Master sequencer state machine. The transitions to various states will be discussed in greater detail following the state machine.

The attached equations (Appendix A) listing should be used as reference along with the Master Sequencer State Machine diagram in order to interpret the following state machine logic description.

The machine is in IDLE state when there are no requests for a bus transfer. On a processor PCI transaction request (generated by decoder, included in design), and the PCI bus grant from the external arbiter, the state machine transitions to the ADDR state. The PCI specification requires that there be only one central arbiter in the PCI system. This design assumes that the arbiter is implemented off board. If the processor is using address stepping, then the transition is to the DR_BUS state from the IDLE state.

Once in ADDR state, on the next clock the DATA state is reached unconditionally. In the ADDR state the appropriate command bus signals are driven. These define the PCI bus command, for example, 0010 specifies an I/O read cycle. These are generated from the processor read/write, IO/memory and data/code signals, which are

used by the i486 to define the processor cycle. FRAME, which signals the start of a PCI cycle, is generated in the ADDR state and is held active through the DATA state till the Target/processor asserts a cycle complete signal.

In the DATA state, data is transferred from the Master to Target in case of a write, or from the Target to Master in case of a read. Wait states can be added by the Target by asserting TRDY or by Master by deasserting IRDY. In case of a read cycle, a turnaround cycle is required between the ADDR and DATA phases in order to avoid contention when one agent stops driving the signal and another agent starts driving. The turnaround wait state is asserted by the Target. (See PCI read cycle timing diagram, Figure 11 and Appendix A.). The DATA and DATA1 state are identical, the DATA1 state is needed for parity purposes.

In case of fast back to back processor cycles, the machine remains in the DATA1 state. A flag SA is used to determine if the current PCI cycle is going to the same Target as the previous cycle. Flag L_CYC is set when the current cycle is a write and the previous cycle was also a write. These flags determine the presence of fast back to back cycles. The state machine transitions to TURN_AR state if the cycles are not back to back, in preparation for completing the cycle and tri-stating the bus signals. If the Target asserts a STOP (stop current cycle), the machine transitions from the DATA1 state to the S_TAR state.

The DR_BUS state is needed only if address stepping is used. In this design, transitions to this state are used for the Master to park on the PCI bus, while the processor is stepping through a cycle.

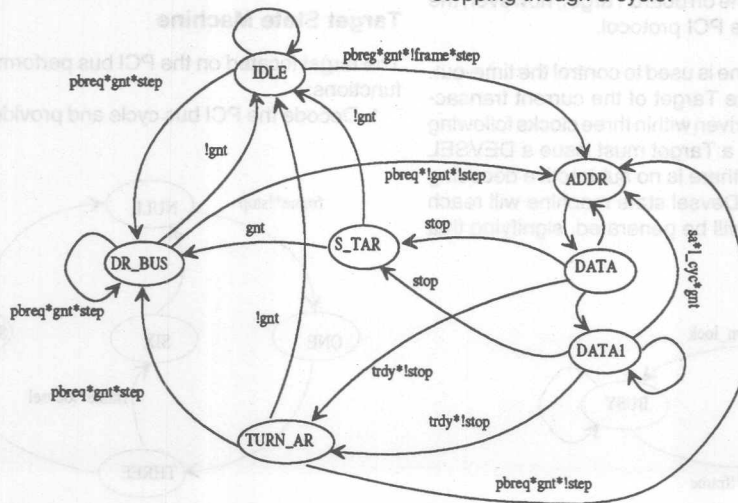


Figure 10. Master Sequencer State machine

PCI Bus Implementation

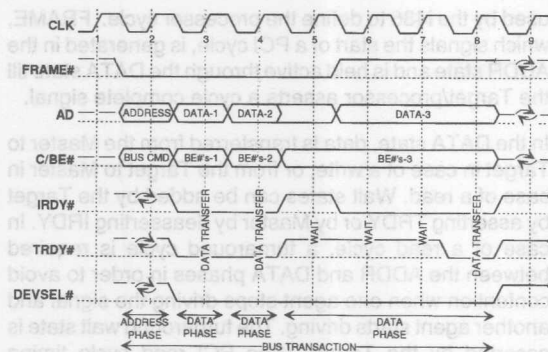


Figure 11. PCI Bus Read Cycle (from PCI spec.)

Finally, in the S_TAR state, if grant is valid, the machine transitions to DR_BUS state.

PCI provides an access mechanism which allows non-exclusive access to processors in the face of an exclusive access. This is referred to resource lock. This mechanism is based on locking only the PCI resource to which the original locked access was Targeted. The LOCK signal indicates that an exclusive lock is underway. The Master state machine controls the master lock mechanism. It has only 2 states, BUSY and FREE. The FREE state implies that the bus is not locked by any Master or the current Master has it locked. If another Master owns the lock, the state transitions to BUSY and stays there till LOCK and FRAME are deasserted. The LOCK state machine has not been simulated, since resource locks are not implemented in the on board Target, however, the equations are as per the PCI protocol.

The Devsel State machine is used to control the time-out. DEVSEL is driven by the Target of the current transaction. DEVSEL must be driven within three clocks following the address phase, i.e., a Target must issue a DEVSEL before any response. If there is no subtractive decoding in the system, then the Devsel state machine will reach state SIX and time-out will be generated, signifying that

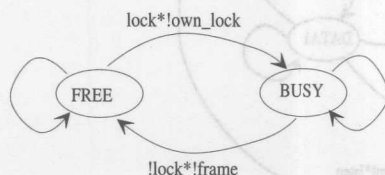


Figure 13. Master Lock State machine

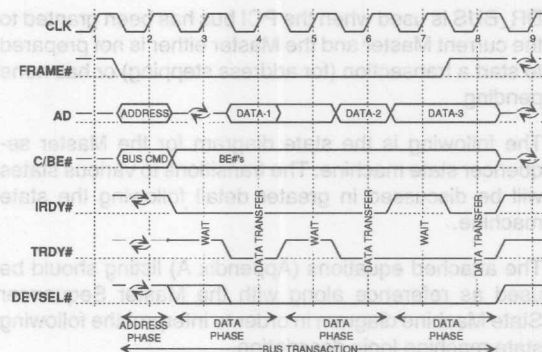


Figure 12. PCI Bus Write Cycle (from PCI spec.)

no Target decoded the address. This will enable the Master to terminate the transaction.

In addition to the above state machines, the Master Controller has a five bit counter, which runs on a 1 MHz clock. This counter is used to generate the MAS_TO signal. This provides a 32 micro seconds latency for all PCI transactions. Latency is defined as the time from when FRAME is asserted to TRDY being asserted. Typical latencies are relatively short, however worst case latencies may be quite long and unpredictable, for example, latency to a standard expansion adapter (ISA/EISA) through a bridge is often a function of the adapter behavior, not PCI behavior. The length of the latency time-out can be modified In-System as desired for a low latency system.

Target State Machine

The target located on the PCI bus performs the following functions:

1. Decode the PCI bus cycle and provide data during a

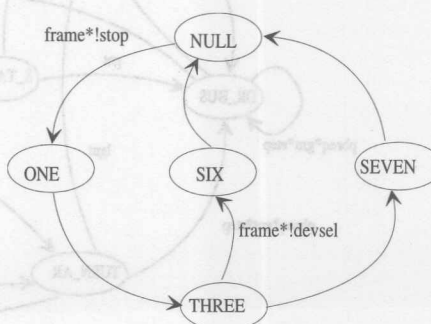


Figure 14. Devsel State Machine

- read
2. Generate parity on PCI bus
3. Generate target abort to terminate a bus cycle
4. Insert wait state during a read cycle between address and data phase

The PCI specification requires that the Target state machine be independent of the Master state machine. The Target interface has a backend that is responsible for determining when a transaction is terminated. The location of the Target in the Master backend address space can be changed In-System. Furthermore, subtractive decoding can be introduced if desired. This will make sure that the DEVSEL time-out is never asserted. The backend can also implement a resource lock. In this design, resource locks are not included in the target and zero wait state address decoding is assumed. The protocol for the target is fairly simplistic. The Master asserts the address, on a read cycle, if the target has a address hit, it initiates its internal state machine and either supplies the data or asserts an abort signal. Following is the Target state machine state description:

TGT_IDLE: In this state the machine is waiting for a decode to the target, i.e., the on board decoder sees a bus cycle directed to the target. The machine transitions to TGT_DATA on HIT. This path can be removed if the Target cannot do single cycle decodes. If STOP is asserted by the Target, the machine transitions to BACKOFF. The machine goes to state B_BUSY when it sees FRAME asserted on the bus, but the HIT signal is still invalid.

B_BUSY: The Target waits for the current transaction to complete and the bus to return to idle. This state is useful for devices that do slow address decode or perform subtractive decode. In this design, both these are not supported, hence there is no transition to the TGT_DATA and BACKOFF states.

TGT_DATA: The Target transfers data in this state. The machine transitions to BACKOFF if FRAME and STOP are asserted. In case of read cycle, the target asserts a wait state after the address is driven on the bus by the processor. This wait state is asserted by delaying the assertion of TRDY.

BACKOFF: The target goes to this state after it asserts STOP and waits for the Master to deassert FRAME.

TURN: This state is reached when the transaction is completed. In preparation for the bus signals to be tristated.

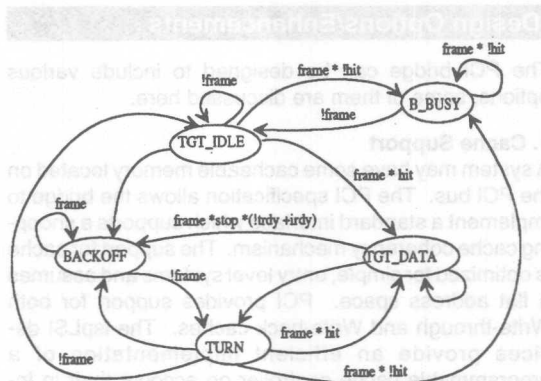


Figure 15. Target State Machine

In addition to the above state machine the Target also contains a trivial command bus state machine. This machine is responsible for storing the command bus information during the address phase of the bus cycle. This is required since the command bus carries the byte enables during the data phase and the cycle type information is lost.

Parity

PCI compliant devices are required to implement parity control. PCI bus has two signals, PAR and PERR that driven by the Master or Target. PAR is used to drive an even parity, covering AD3..AD0 and C/BE3..C/BE0, during address and data phases. To ensure the correct bus operation is performed, the four command lines are included in the parity calculation. In this design, parity generation is supported. The i486 processor drives DP0..DP3 lines which contain the parity bits for the 4 bytes of the processor bus. These bits and the data/address lines are used to generate PAR. The Lattice ispLSI device has a hardware 8-input XOR that can be used for this purpose. The Master drives the PAR onto the PCI bus during a write cycle. The Target is responsible for driving the PERR signal during the write cycle, if it has a parity error. During a read cycle, the Master generates the PERR based on the PCHK signal provided by the i486 processor. The Master also generates the PAR signal based on the state of PAR which is asserted by the Target in the read cycle. The PAR signal is generated by the Target on a read cycle. This design does not incorporate this feature, however it can be implemented quite nicely in an additional ispLSI device, since all the AD lines and the local processor lines are needed for generating the PAR bit.

PCI Bus Implementation

Design Options/Enhancements

The PCI bridge can be designed to include various options, some of them are discussed here.

1. Cache Support

A system may have some cacheable memory located on the PCI bus. The PCI specification allows the bridge to implement a standard interface which supports a snooping cache coherency mechanism. The support for cache is optimized for simple, entry level systems and assumes a flat address space. PCI provides support for both Write-through and Write-back caches. The ispLSI devices provide an efficient implementation of a programmable cache controller on account their in In-System Programmability, which makes the design flexible to support various cache schemes.

2. 64 Bit Data Bus

PCI provides a 64 bit extension to the data bus for agents with a 64 bit data bus. This requires 39 additional pins: REQ64, ACK64, PAR64, C/BE4..C/BE7 and AD32..AD63. Basically the 64 bit bus works the same way as the 32 bit bus. In this design, the data lines are not driven by the ispLSI device, which actually drives the control signals to enable the external buffers. This would make the expansion to 64 bit mode real simple. The internal logic can be modified to support the additional control signals. REQ64 is the pin used by the Master to request a 64 bit transfer. This is an extremely attractive option for 64 bit processors such as Pentium. When implementing this option, one has to be careful since Double Word swapping is allowed on the 64 bit data bus.

3. 64 Bit Addressing

PCI supports addressing beyond 4GB by defining a mechanism to transfer a 64 bit address from the Master to Target. A 64 bit address can be provided in one clock if the 64 bit address/data bus is being used. The Dual Address Cycle mode can be used, for 32 bit systems where the address is transferred in two clocks. This option cannot support address stepping on account of the two clock address transfer.

4. Slow Decoding Targets/Subtractive Decoding

This design assumes that the target can decode the PCI bus address with no wait state. For slower Targets, additional transitions can be added into the Target state machine, namely, transition from B_BUSY state to TGT_DATA and BACKOFF can be added. In addition, the path from IDLE to TGT_DATA can be removed if the Target cannot do single cycle decoding. Additional logic will depend on the specific Target implementation.

Other design options would be to include interrupt generation or even implement the entire interrupt controller in the master interface for PCI as well as local interrupts. Target Resource lock is another viable option. In a resource lock, exclusivity of an access is guaranteed by the target of the access, not by excluding all other accesses. This allows future processors to hold a hardware lock across several accesses without interfering with non-exclusive accesses such as video.

Conclusion

This application note has presented a broad overview of the PCI bus along with a sample PCI Master/Target interface implemented in a Lattice ispLSI device. With the popularity of the non-proprietary, high performance and extremely flexible local bus, it is not surprising that designers are looking to programmable logic to meet the challenges offered by a PCI interface design. The Lattice In-System Programmable device family is ideally suited to such complex state machine intensive applications. While the sample design in this application note is specific enough to cover the required PCI protocol, it is adaptable and can be molded around any given peripheral or processor. In fact, it can even be reconfigured in the system from one peripheral to another, as long as the hardware interface is not too rigid. Additional features can always be added either in more ispLSI devices or discrete logic on account of the modular layout of the design.

The source file for the design is included in the following pages. This design is implemented using ABEL 4.1.3 software with Lattice pDS+ ABEL Fitter. pLSI Property Statements provide the user direct control over hardware specific features of the ispLSI and pLSI devices. The simulations were carried out using Viewlogic ViewSim software. Alternatively, the design can be implemented quite easily using the Lattice pDS+ Development System.

References

1. PCI Local Bus Specification, Rev. 2.0
2. Lattice Data Book, 1994
3. Lattice pDS+ Software, User Manual.
4. ABEL Design Software, User Manual.

Design Equations and Simulation Waveform

```
module pci_master
title 'pci bus master interface for i486 cpu';

"NOTES:
"this design assumes that there is no cacheable memory located
"as a target on the pci bus
"This design is a guideline for implementing PCI bus bridge
"for a 486 cpu interface. This design does not implement a 100%
"PCI compatible bridge, however, the basic state machine is
"implemented and provides a baseline to build a complete PCI
"master interface
```

```
*****
*****
***** plsi properties *****
*****
*****
*****
```

```
plsi property 'timing_sim pci_mast';
plsi property 'strong_route extended';
plsi property 'try 4';
plsi property 'max_delay 1';
```

```
*****
*****
***** declarations *****
*****
*****
*****
```

```
pci_master device 'p1032j09';
```

```
*****
***** inputs *****
*****
***** inputs for processor interface *****
```

```
pa0 pin; "processor address lines
pa1 pin; "processor address lines
pa2 pin; "processor address lines
pa3 pin; "processor address lines
pa4 pin; "processor address lines
pa5 pin; "processor address lines
pa6 pin; "processor address lines
pa7 pin; "processor address lines
pa8 pin; "processor address lines
pa9 pin; "processor address lines
pa10 pin; "processor address lines
pa11 pin; "processor address lines
pa12 pin; "processor address lines
pa13 pin; "processor address lines
pa14 pin; "processor address lines
pa15 pin; "processor address lines
pa16 pin; "processor address lines
pa17 pin; "processor address lines
pa18 pin; "processor address lines
pa19 pin; "processor address lines
pa20 pin; "processor address lines
pa21 pin; "processor address lines
pa22 pin; "processor address lines
pa23 pin; "processor address lines
pa24 pin; "processor address lines
pa25 pin; "processor address lines
pa26 pin; "processor address lines
```

PCI Bus Implementation

```

pa27 pin;          "processor address lines
pa28 pin;          "processor address lines
pa29 pin;          "processor address lines
pa30 pin;          "processor address lines
pa31 pin;          "processor address lines
pbe0 pin;          "processor byte enables
pbe1 pin;          "processor byte enables
pbe2 pin;          "processor byte enables
pbe3 pin;          "processor byte enables
!plock pin;        "processor lock pin
!pdata pin;        "processor C/D pin
!piom pin;         "processor IO/m pin
!pbreq pin;        "processor bus request
!pread pin;        "processor read/write
dp0,dp1,dp2,dp3 pin; "processor parity pins

step pin;          "stepping input for debugging
cclk pin;          "clock - 1mhz
pclk pin;          "clock for timeout counter

"master input pins from pci
!gnt pin;          "from bus arbiter
!trdy pin;         "from target
!stop pin;         "from target
!devsel pin;       "indicates tgt has been selected
ready pin;         "indicates ready to transfer
comp pin;

par pin;          "bidirectional parity pin

"slave inputs
term pin;          "slave wants to terminate the bus cycle
tar_dly pin;

"*****
"          outputs
"*****
"master output pins and bi directionals

cbe0 pin;          "processor address lines
cbe1 pin;          "processor address lines
cbe2 pin;          "processor address lines
cbe3 pin;          "processor address lines
data_en pin;       " enables the data buffers on the pCI bus

!frame pin;        "processor address lines
!lock pin;         "processor address lines
!req pin;          "processor address lines
!irdy pin;         "processor address lines
addr_en pin;       "processor address lines
mas_abort pin;     "transaction aborted by master due to timeout

"The following is the output enable for the external buffers
ad_oe pin;

"*****
"          nodes
"*****
mas_to node;       "internal timer has expired
pci node;          "cpu access is on pci bus from built in address decoder
dev_to node;       "devsel timeout on pci bus, ie, DEVSEL was not asserted
sa node;           "last cycle to same tgt as current
L_cyc node;        "last cyc was a write, bit set in register
Own_lock node;     "master owns lock

```

```

tgt_abort node;          "target aborts access
tgtl node;
tgtlr node istype 'buffer,reg_d';  "used to store tgt access info.
ldt pin;
preadr node istype 'buffer,reg_d';  "used to store write/read cycle bit

"target related nodes
hit node;
cmdr3,cmdr2,cmdr1,cmdr0 node;

"***** Other Definitions *****"
"***** TARGET MACHINE DEFN. *****"
"defn. of all bus cycles
int_ack          = [0,0,0,0];
spec_cyc         = [0,0,0,1];
io_read          = [0,0,1,0];
io_write         = [0,0,1,1];
res1             = [0,1,0,0];  "RESERVED
res2             = [0,1,0,1];  "RESERVED
mem_read         = [0,1,1,0];
mem_write        = [0,1,1,1];
res3             = [1,0,0,0];  "RESERVED
res4             = [1,0,0,1];  "RESERVED
config_read      = [1,0,1,0];
config_write     = [1,0,1,1];
mem_wr_mult      = [1,1,0,0];
dual_add_cyc     = [1,1,0,1];
"for 64 bit addressing only- not supported by this design
mem_read_line    = [1,1,1,0];
mem_wr_inval     = [1,1,1,1];

cmd = [cbe3..cbe0];  " for convenient definition of cbeX lines used in 1st phase
of bus cycle
cmdr = [cmdr3..cmdr0];  "storage for command bus
pbex = [pbe3..pbe0];  "processor byte enables

"***** MASTER MACHINE DEFN. *****"
"master lock machine
lreg node;
lreg istype 'buffer,reg_d';

"state definitions for lock machine
free = 0;
busy = 1;

"devsel timer machine
d2,d1,d0 node;
dreg = [d2..d0];
dreg istype 'buffer,reg_d';

"state defn. for devsel state machine
null = [0,0,0];
one = [0,0,1];
three = [0,1,1];
seven = [1,1,1];
six = [1,1,0];

"master sequencer machine
s0,s1,s2 node;
sreg = [s2..s0];
sreg istype 'buffer,reg_d';

"state defn. for master sequencer machine

```

PCI Bus Implementation

```

idle = [0,0,0] ;
addr = [0,0,1] ;
data = [0,1,1];
data1 = [1,0,1]; " for parity purpose
dr_bus = [1,1,1];
turn_ar = [1,1,0];
s_tar = [1,0,0];

"counter defn.
q0,q4,q3,q2,q1 node;
count = [q4..q0];
count istype 'buffer,reg_d';

"***** TARGET MACHINE DEFN.*****
t2,t1,t0 node;
treg = [t2..t0];
treg istype 'reg_d,buffer';

tgt_idle = [0,0,0];
backoff = [0,0,1];
b_busy = [0,1,0];
tgt_data = [0,1,1];
turn = [1,0,1];

"state machine to clock comand bus for target
c1,c0 node;
creg = [c1,c0];
creg istype 'reg_d, buffer';

no_ack = [0,0];
strobe = [0,1];
hold = [1,1];

"***** for convenient definition of opex lines used in lab phase *****
" State machines
*
"*****
"state diagram for sequencer machine
state_diagram sreg;

state idle: "idle state on the bus
if (pbreq & gnt & !frame & !lrdy & !step) "cpu has a pci bus request and address strobe
then addr;

else if ((!pbreq & gnt) # (pbreq & gnt & step)) & (!frame & !lrdy) "park on
bus if stepping
then dr_bus;

else idle;

state addr: "master starts a transaction
goto data;

state data: "master transfers data first data phase
if (frame) # ((!frame & !lrdy & !trdy.pin & !stop.pin & !dev_to) & !(cmd==spec_cyc) & comp))
then data1;

else if (!frame & !step & trdy.pin & !stop.pin & !(cmd==spec_cyc) & sa & L_cyc & pbreq & gnt)
then addr; " only if fast back to back cycles are supported

else if (!frame & trdy.pin & !stop.pin & !(sa & L_cyc & pbreq & gnt) #

```

PCI Bus Implementation

```

if (! (cmd==spec_cyc) & comp))
    then turn_ar;          "turnaround state if no back-back cycles

else if (!frame & stop.pin # !frame & dev_to & !trdy.pin)
    then s_tar;

state datal:
if (frame) # (!frame & !lrdy & !trdy.pin & !stop & !dev_to) & ! (cmd==spec_cyc) &
comp))
    then datal;

    if (!frame & !step & trdy.pin & !stop.pin & ! (cmd==spec_cyc) & sa & L_cyc &
pbreq & gnt)
        then addr;          " only if fast back to back cycles are supported

    else if (!frame & trdy.pin & !stop.pin & ! (sa & L_cyc & pbreq & gnt) #
(! (cmd==spec_cyc) & comp))
        then turn_ar;          "turnaround state if no back-back cycles

    else if (!frame & stop.pin # !frame & dev_to & !trdy.pin)
        then s_tar;

state turn_ar:          " state for houskeeping purposes
if (pbreq & gnt & !step)
    then addr;

    else if (!pbreq & gnt # pbreq & gnt & step)
        then dr_bus;

    else if (!gnt)
        then idle;

    else turn_ar;

state s_tar:          " turnaround state when stop is asserted
if (gnt)
    then dr_bus;

    else if (!gnt)
        then idle;

    else s_tar;

state dr_bus:          "bus parked or address stepping is used
if (pbreq & gnt & step # !pbreq & gnt)
    then dr_bus;

    else if (pbreq & !gnt & !step)
        then addr;

    else if (!gnt)
        then idle;

    else dr_bus;

***** end of master sequencer state machine*****

"state diagram for LOCK machine
state_diagram lreg;

state free:          "bus is locked by current master
if (!lock # lock & Own_lock)
    then free;

```


PCI Bus Implementation

```
else if (lock & !Own_lock)
    then busy;

state busy:
    if (!lock & !frame)
        then free;
    else if (lock & frame)
        then busy;

"***** end of master lock state machine*****"

state_diagram dreg;

state null:
    if (frame & !stop)
        then one;
    else null;

state one:
    goto three;

state three:
    goto seven;

state seven:
    if (!devsel & frame)
        then six;
    else null;

state six:
    goto null;

"***** end of devsel state machine*****"

" ***** target state machine *****"
state_diagram treg;

state tgt_idle:
    if (!frame.pin)
        then tgt_idle;
    else if (frame.pin & !hit)
        then b_busy;
    else if (frame.pin & hit & (!term # term & ready))
        then tgt_data;
    else if (frame.pin & hit & term & !ready)
        then backoff;
    else tgt_idle;

state b_busy:
    if ((frame.pin # irdy.pin) & !hit)
        then b_busy;
    else if (!frame.pin)
        then tgt_idle;
    else b_busy;

state tgt_data:
    if (frame.pin & stop & trdy & !irdy.pin # frame.pin & !stop # !frame.pin & !trdy &
```

```

!stop)
    then tgt_data;

else if (frame.pin & stop & (!trdy # irdy.pin))
    then backoff;

    else if (!frame.pin & (stop # trdy))
        then turn;

-00000000 else tgt_data;

state backoff:
if (frame.pin)
    then backoff;
    else if (!frame.pin)
        then turn;

state turn:
if (!frame.pin)
    then tgt_idle;

    else if (frame.pin & !hit)
        then b_busy;

    else if ( frame.pin & hit & (!term # term & ready))
        then tgt_data;

        else if (frame.pin & hit & (term & !ready))
            then backoff;
"***** end of target state machine*****"
" cmd bus store state machine
state_diagram creg;

state no_ack:
if (!frame.pin)
    then no_ack;

    else if (frame & hit)
        then strobe;

state strobe:
goto hold;

state hold:
if (frame.pin)
    then hold;

    else if (!frame.pin)
        then no_ack;
"*****
equations
lreg.c = pclk;
dreg.c = pclk;
sreg.c = pclk;
treg.c = pclk;
creg.c = pclk;

count.c = cclk;
count.re = trdy & gnt;

"5 bit counter initiated by asserting frame, runs on a 1Mhz clock

```

"reset counter when trdy is generated by master

PCI Bus Implementation

```

"will generate mas_to signal at end of count
q0.d = q0 $ frame;
q1.d = (q0 & frame) $ q1;
q2.d = (q0 & q1 & frame) $ q2;
q3.d = (q0 & q1 & q2 & frame) $ q3;
q4.d = (q0 & q1 & q3 & q3 & frame) $ q4;
mas_to = (q1 & q2 & q3 & q4 & q0 & frame); "master timed out

pci = (pa31 & pa30 & pa29 & pa28 & pbreq); " decoded pci address space f0000000-
ffffff

Own_lock = !lock & !frame & !irdy & pbreq & gnt & plock # Own_lock & (frame # lock);

frame = (sreg==addr) # ((sreg==data)#(sreg==data1) & !dev_to & ((!comp & (!mas_to #
gnt) & !stop.pin) # !ready));

lock = !((Own_lock & (sreg==addr)) # tgt_abort # dev_to #
((sreg==data)#(sreg==data1)) & stop.pin & !trdy.pin & !ldt)
# (Own_lock & !plock & comp & ((sreg==data)#(sreg==data1)) & !frame &
trdy.pin));

req = (pbreq & !plock # pbreq & plock & (lreg==free)) & !(sreg==s_tar);

irdy = ((sreg==data)#(sreg==data1)) & (ready # dev_to);

dev_to = (dreg==six);

mas_abort = mas_to;

cmd = (int_ack & ((sreg==addr) & pread & piom & pdata)
# io_read & ((sreg==addr) & pread & piom & !pdata)
# io_write & ((sreg==addr) & !pread & piom & !pdata)
# mem_read & ((sreg==addr) & pread & !piom & !pdata)
# mem_write & ((sreg==addr) & !pread & !piom & !pdata)
# spec_cyc & ((sreg==addr) & !pread & piom & pdata)
# pbex & (sreg==data)
# pbex & (sreg==data1)
# pbex & ((sreg==dr_bus) & step & pbreq));

addr_en = (sreg==addr);

data_en = (sreg==data)#(sreg==data1)#(sreg==dr_bus);

"preadr is used to store the write/read access
preadr.d = pread & gnt;

preadr.clk = pclk;

preadr.ar = (!gnt & !pci);

L_cyc = !pread & preadr.q;

tgt_abort = (stop.pin & !devsel.pin & ((sreg==data)#(sreg==data1)) & !frame & irdy);

"the following equations assume only 1 target device. The access to the device
"is stored for back to back transfers. this can be expanded to include more devices

tgt1 = (pbreq & pa31 & pa30 & pa29 & pa28 & pa27 & pa26 & pa25 & pa24 ); "FF000000-
FFFFFFF

tgt1r.d = tgt1;

tgt1r.ar = (!gnt & !pci); "reset the register when there is a non-pci access

```

```

tgtlr.c = pclk;

sa = tgtlr.q & tgtl;

"***** output enables*****

cmd.oe = (sreg==addr) # (sreg==data) # (sreg==dr_bus) # (sreg==data1);

lock.oe = Own_lock & ((sreg==data)#(sreg==data1)) # (lock.oe & (frame # lock));

ad_oe = (sreg==addr) # (sreg==dr_bus) & step & pbreq;

"irdy needs to be asserted when addr or data are the previous states
irdy.oe = (sreg==addr)
          #((sreg==idle) & pbreq & gnt & !frame & !irdy & pci & !step) "asserted
when addr is next state
          #((sreg==turn_ar) & pbreq & gnt & !step) "asserted when addr is
next state
          #((sreg==data) & !frame & !step & trdy.pin & !stop.pin & !(cmd==spec_cyc)
& sa & L_cyc & pbreq & gnt)
          #((sreg==dr_bus) & pbreq & !gnt & !step) "asserted when addr is
next state
          #(((sreg==data) & frame) # ((!frame & !irdy & !trdy.pin & !stop.pin &
!dev_to) & !(cmd==spec_cyc) & comp)))
          #(((sreg==data1) & frame) # ((!frame & !irdy & !trdy.pin & !stop.pin &
!dev_to) & !(cmd==spec_cyc) & comp)));

"*****
" Parity logic
"*****
par = ((dp0 $ dp1) $ (dp2 $ dp3));
" # (from slave par circuit);

par.oe = (sreg==data) & (cmd==io_write) # (cmd==mem_write) "for address parity
          # (sreg==data1) & (cmd==io_write) # (cmd==mem_write) "for data parity
          # (treg==tgt_data) & trdy & ((cmdr==io_read) # (cmdr==mem_read)); " for
slave driven par

"***** target equations*****
trdy = !(ready & !tgt_abort & (treg==tgt_data)
          & (((cmdr==io_write) # (cmdr==mem_write))
          # ((cmdr==io_read) # (cmdr==mem_read) & tar_dly)));

stop = !((treg==backoff) # (treg==tgt_data) & (tgt_abort # term)
          & (((cmdr==io_write) # (cmdr==mem_write))
          # ((cmdr==io_read) # (cmdr==mem_read) & tar_dly)));

devsel = !((treg==backoff) # (treg==tgt_data) & !tgt_abort);

"perr = (from parity circuit)

trdy.oe = (treg==backoff) # (treg==tgt_data) # (treg==turn);

stop.oe = (treg==backoff) # (treg==tgt_data) # (treg==turn);

devsel.oe = (treg==backoff) # (treg==tgt_data) # (treg==turn);

"hit = (decode of PCI address lines );

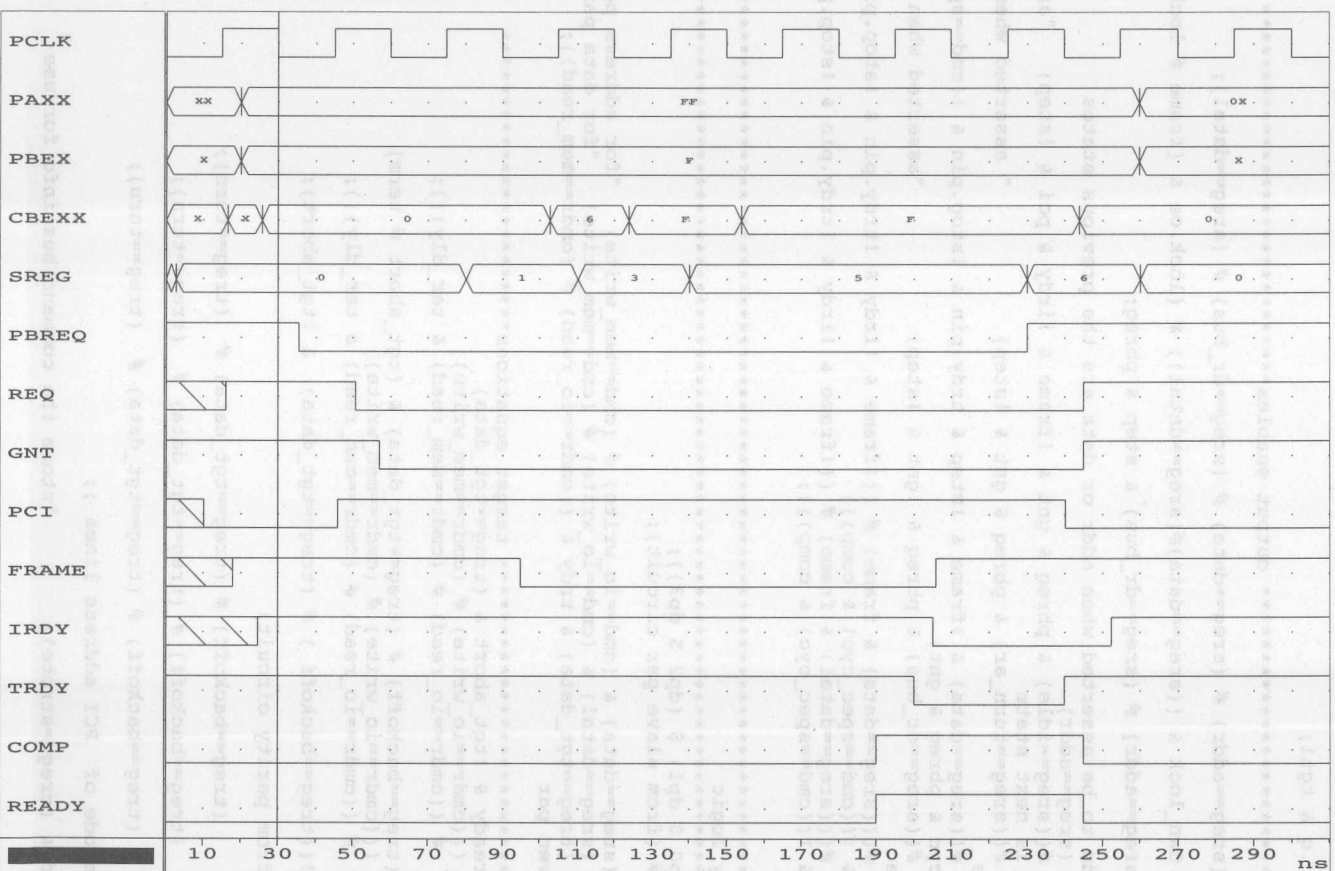
cmdr = cmd & (creg==strobe); "store the command bus info for use

END;

```

PCI Bus Implementation

PCI Bus Read Cycle (Simulation)



pDS+ Fitter Report

```
*****
*                               *
*      Lattice pDS+ Fitter Report      *
*                               *
*****
```

Copyright (c) Lattice Semiconductor Corp. 1992. All Rights Reserved.

```
Design Name: pci_mast
File: pci_mast.doc
Date/Time: Mon Apr 25 12:13:33 1994
Targeted Device: pLSI1032-90LJ84
Software Version: DPM 1.60 12/8/93
```

Fitter Parameters Used

```
-----
AVG_GLB_IN: 16
EFFORT: 4
IGNORE_FIXED_PIN: OFF
MAX_DELAY: 1
MAX_GLB_IN: 16
PARAM_FILE: (null)
PART: pLSI1032-90LJ84
TIMING_SIM: pci_mast
TRY: 4
FAST_ROUTE: OFF
STRONG_ROUTE: EXTENDED
```

Process Status

```
-----
Design Analysis: complete
Logic Partitioning: complete
Place and Route: complete
Post Route: complete
Fuse File Generation: complete
Merging TMV in JEDEC: incomplete
```

```
*****
*                               *
*      Post-Route Report      *
*                               *
*****
```

```
Design Name: pci_mast
Targeted Device: pLSI1032-90LJ84
Date/Time: Mon Apr 25 13:09:06 1994
```

Software Version: 1.00.35

All strategy results:
 Strategy 4 - Estimated No. of GLBs : 19
 Strategy 4 - Estimated No. of GLB Levels: 3

Final Selected Strategy 4 - Estimated No. of GLBs : 19
 Strategy 4 - Estimated No. of GLB Levels: 3

Partitioning:

Total number of GLBs : 31

PCI Bus Implementation

Total number of Product Terms used : 193
Average number of Product Terms : 6.2
Total number of nets created : 119
Average number of Inputs per GLB : 9.6
Average number of Outputs per GLB : 2.2
Number of I/Os Generated : 46
Number of Dedicated Inputs Generated : 4
Type of Clocks Generated : 2 System Clocks
: 0 I/O Clocks
: 0 Product Term Clocks

qds+ Filter Report

Lattice qds+ Filter Report

Copyright (c) Lattice Semiconductor Corp. All Rights Reserved.
Design Name: pci_mast
File: pci_mast.doc
Date/Time: Mon Apr 25 13:13:33 1994
Targeted Device: PL811032-30LJ84
Software Version: DPM 1.00 12/8/93

Filter Parameters Used

STRONG ROUTE: EXTENDED
FAST ROUTE: OFF
TRY: 4
TIMING_CIN: pci_mast
PART: PL811032-30LJ84
PARAM FILE: (null)
MAX_GLB_IN: 16
MAX_DELAY: 1
IGNORE_FIXED_PIN: OFF
ERROR: 4
AVE_GLB_IN: 16

Process Status

Merging TMV in JEDBC: incomplete
Parse File Generation: complete
Post Route: complete
Place and Route: complete
Logic Partitioning: complete
Design Analysis: complete

Post-Route Report

Design Name: pci_mast
Targeted Device: PL811032-30LJ84
Date/Time: Mon Apr 25 13:09:02 1994
Software Version: 1.00.32

All strategy results:
Strategy 4 - Estimated No. of GLBs : 19
Strategy 4 - Estimated No. of GLB Levels : 3
Final Selected Strategy 4 - Estimated No. of GLBs : 19
Strategy 4 - Estimated No. of GLB Levels : 3

Partitioning:

Total number of GLBs : 31

PCI Bus Implementation

Post Route Pin Report

Post-Route Pin Report

Pin Number	Signal Name	Fixed	Pin Type
1	GND	Yes	Gnd
3	mas_abort	No	Output
5	pa29	No	Input
6	frame	No	Output
7	irdy-	No	Input
9	devsel-	No	Input
10	irdy	No	Bidi
11	par	No	Output
14	comp	No	Input
16	pa28	No	Input
20	pclk	Yes	Clock
21	VCC	Yes	Vcc
22	GND	Yes	Gnd
25	pbe2	No	Input
26	hit	No	Input
27	cbe3	No	Bidi
28	cbe2	No	Bidi
29	req	No	Output
30	addr_en	No	Output
31	term	No	Input
32	tar_dly	No	Input
33	frame-	No	Input
34	pa31	No	Input
35	devsel	No	Bidi
37	pa27	No	Input
38	dp0	No	Input
39	data_en	No	Output
40	stop	No	Bidi
41	pa26	No	Input
42	pbe3	No	Input
43	GND	Yes	Gnd
44	pbe1	No	Input
45	cbe0	No	Bidi
46	gnt	No	Input
47	pdata	No	Input
48	cbe1	No	Bidi
49	trdy-	No	Input
50	dp3	No	Input
51	dp2	No	Input
52	dpl	No	Input
54	pread	No	Input
55	plock	No	Input
56	ad_oe	No	Output
64	GND	Yes	Gnd
65	VCC	Yes	Vcc
66	cclk	Yes	Clock
68	pbreq	No	Input
69	pbe0	No	Input
70	ready	No	Input
71	piom	No	Input
72	pa24	No	Input
77	#lock	No	Bidi
78	step	No	Input
79	pa30	No	Input
81	stop-	No	Input
82	pa25	No	Input
83	trdy	No	Bidi
84	ldt	No	Input

Notes

Post Route Pin Report

Post-Route Pin Report

Pin Number	Signal Name	Fixed	Pin Type
1	GND	Yes	Out
2	max_addr	No	Out
3	pa23	No	Input
4	frame	No	Out
5	lady-	No	Input
6	device-	No	Input
7	lady	No	Input
8	par	No	Out
9	comp	No	Input
10	pa23	No	Input
11	colx	Yes	Out
12	VCC	Yes	Out
13	GND	Yes	Out
14	pa23	No	Input
15	lady	No	Input
16	pa23	No	Input
17	lady	No	Input
18	pa23	No	Input
19	lady	No	Input
20	pa23	No	Input
21	lady	No	Input
22	pa23	No	Input
23	lady	No	Input
24	pa23	No	Input
25	lady	No	Input
26	pa23	No	Input
27	lady	No	Input
28	pa23	No	Input
29	lady	No	Input
30	pa23	No	Input
31	lady	No	Input
32	pa23	No	Input
33	lady	No	Input
34	pa23	No	Input
35	lady	No	Input
36	pa23	No	Input
37	lady	No	Input
38	pa23	No	Input
39	lady	No	Input
40	pa23	No	Input
41	lady	No	Input
42	pa23	No	Input
43	lady	No	Input
44	pa23	No	Input
45	lady	No	Input
46	pa23	No	Input
47	lady	No	Input
48	pa23	No	Input
49	lady	No	Input
50	pa23	No	Input
51	lady	No	Input
52	pa23	No	Input
53	lady	No	Input
54	pa23	No	Input
55	lady	No	Input
56	pa23	No	Input
57	lady	No	Input
58	pa23	No	Input
59	lady	No	Input
60	pa23	No	Input
61	lady	No	Input
62	pa23	No	Input
63	lady	No	Input
64	pa23	No	Input

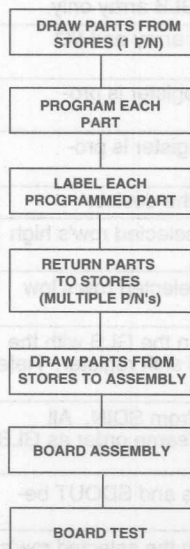
Overview

Increasing demand for high pin count programmable logic and FPGAs creates many manufacturing challenges. These devices can require extra steps in the manufacturing flow due to device programming requirements, marking and storage. The Lattice family avoids these extra steps through its In-System Programming (ISP) interface, allowing the devices to be installed prior to programming, and then programmed by the Automated Test Equipment (ATE), as shown in Table 1.

This reduction in the number of steps results in large cost savings to the manufacturer, and offers other advantages as well:

- no dedicated programmers needed
- programmable parts no longer need to be socketed
- the final product is easily upgraded in the field, reducing maintenance costs

Standard Flow Using Non-ISP Devices



Enhanced Flow Using ISP Devices

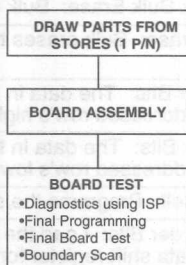


Table 1. Detail of the ISP interface.

ISP interface

The ISP interface is based on a simple 5 signal 5V interface much like the boundary scan chain. The programming of the device is controlled 'on chip' by a simple state machine. Figure 1 illustrates a typical configuration where the programming signals are generated by a generic block called programming control circuitry. The programming process consists of transferring the logic implementation stored in a JEDEC compatible fuse pattern into the device. The method by which the transfer is accomplished is dependent on the end system's definition. The programming control circuitry can be implemented by traditional PLD programmers, IC or printed circuit board testers, the I/O port of a computer such as PC parallel port or a micro controller or microprocessor directly on the system board. In this case we shall concentrate on using the tester as the programming control circuitry.

Device Programming Architecture

The in-system programming of the ispLSI device is controlled by the five programming control interface signals — ispEN, MODE, SCLK, SDI and SDO. The programming information from the JEDEC file is serially shifted into the device via the SDI pin and shifted out through the SDO pin. The ispEN signal controls whether the device is in normal operating mode or programming

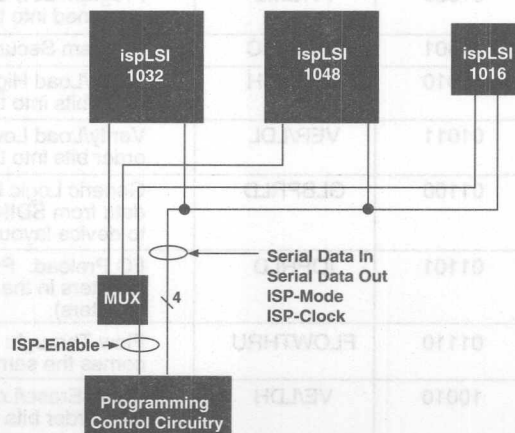


Figure 1. Detail of the ISP Interface

Programming ispLSI Devices with a Tester

mode. SCLK provides the clock to run the state machine. MODE and SDI provide control inputs to the state machine. The internal state machine has a simple instruction set to control the flow of data to either the address registers, data registers or to implement the programming options. Table 2 lists some of the key instructions that can be implemented by the state machine.

The programming circuitry for the ispLSI 1032 is detailed in Figure 2. SDI can drive either the SDO pin, the high order shift register, the low order shift register or the address shift register. The ISP state machine controls where SDI is being driven to and what SDO is being driven by.

The ISP state machine consists of three basic states:

- Idle
- Shift
- Execute

Transition between the state are controlled by the SDI and Mode pins as detailed in Figure 3. The idle state allows the programming to be halted or the device identification data to clocked out of the device. Address, Data, or Command information is shifted in or out of the shift registers while in the shift state. The Execute state is used to execute or 'run' the loaded command. All of the state transitions are controlled by the synchronous clock (SCLK).

The first step when programming the ispLSI devices is to determine what type of device is being programmed.

Table 2. State Machine Instruction Set

Instruction	Operation	Description
00000	NOP	No operation performed
00001	ADDSHFT	Address Register Shift: Shifts address into the address shift register from SDIN.
00010	DATASHFT	Data Register Shift: Shifts data into or out of the data serial shift register.
00011	UBE	User Bulk Erase: Erase the entire device.
00100	GRPBE	Global Routing Pool Bulk Erase: Bulk erases the GRP array only.
00101	GLBBE	Generic Logic Block Bulk Erase: Bulk erases all the GLB array only.
00110	ARCHBE	Architecture Bulk Erase: Bulk erases the architecture array and I/O configuration only.
00111	PRGMH	Program High Order Bits: The data in the data shift register is programmed into the addressed row's high order bits.
01000	PRGML	Program Low Order Bits: The data in the data shift register is programmed into the addressed row's low order bits.
01001	PRGMS	Program Security Cell: Programs the security cell of the device.
01010	VER/LDH	Verify/Load High Order Bits: Load the data from the selected row's high order bits into the data shift register for verification.
01011	VER/LDL	Verify/Load Low Order Bits: Load the data from the selected row's low order bits into the data shift register for verification.
01100	GLBPRLD	Generic Logic Block Preload: Preloads the registers in the GLB with the data from SDIN. All registers in the GLB form a serial shift register. Refer to device layout section for details.
01101	IOPRLD	I/O Preload: Preloads the I/O registers with the data from SDIN. All registers in the I/O cell form a serial shift register (the same order as GLB registers).
01110	FLOWTHRU	Flow Through: Bypasses all the internal shift registers and SDOOUT becomes the same as SDIN.
10010	VE/LDH	Verify Erase/Load High Order Bits: Load the data from the selected row's high order bits into the data shift register for erased verification.
10011	VE/LDL	Verify Erase/Load Low Order Bits: Load the data from the selected row's low order bits into the data shift register for erased verification.

Programming ispLSI Devices with a Tester

Figure 2. Detail of the ispLSI 1032 Programming Architecture

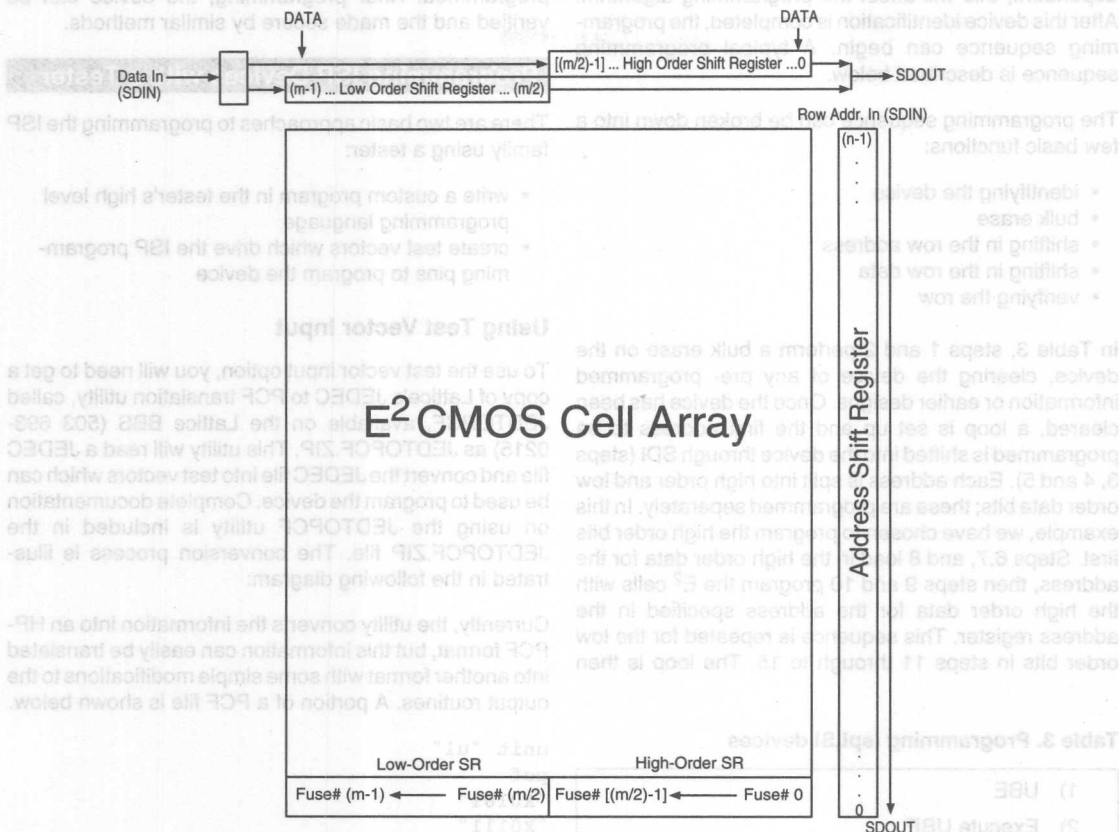
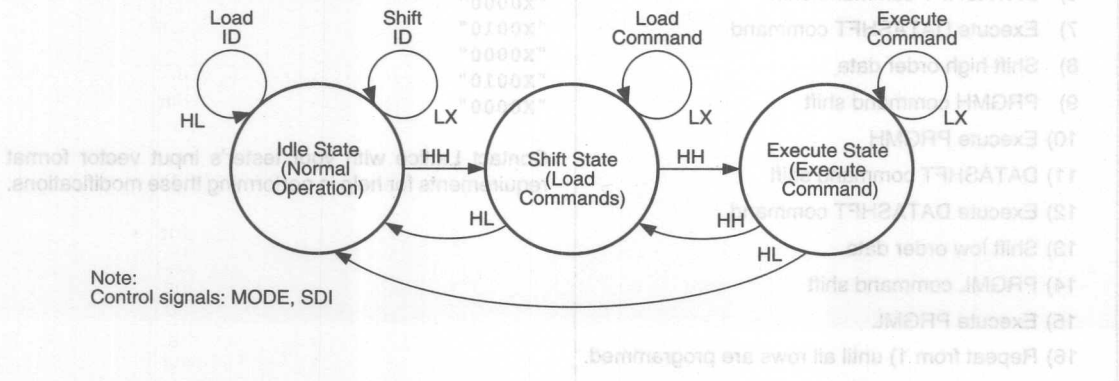


Figure 3. The ISP control state machine.



Programming ispLSI Devices with a Tester

Since the size of the data and address registers is device dependent, this will affect the programming algorithm. After this device identification is completed, the programming sequence can begin. A typical programming sequence is described below.

The programming sequence can be broken down into a few basic functions:

- identifying the device
- bulk erase
- shifting in the row address
- shifting in the row data
- verifying the row

In Table 3, steps 1 and 2 perform a bulk erase on the device, clearing the device of any pre-programmed information or earlier designs. Once the device has been cleared, a loop is set up and the first address to be programmed is shifted into the device through SDI (steps 3, 4 and 5). Each address is split into high order and low order data bits; these are programmed separately. In this example, we have chosen to program the high order bits first. Steps 6, 7, and 8 load in the high order data for the address, then steps 9 and 10 program the E² cells with the high order data for the address specified in the address register. This sequence is repeated for the low order bits in steps 11 through to 15. The loop is then

repeated with the next address until the device is fully programmed. After programming, the device can be verified and the made secure by similar methods.

Programming ISP Devices with a Tester

There are two basic approaches to programming the ISP family using a tester:

- write a custom program in the tester's high level programming language
- create test vectors which drive the ISP programming pins to program the device

Using Test Vector Input

To use the test vector input option, you will need to get a copy of Lattice's JEDEC to PCF translation utility, called JEDTOPCF, available on the Lattice BBS (503 693-0215) as JEDTOPCF.ZIP. This utility will read a JEDEC file and convert the JEDEC file into test vectors which can be used to program the device. Complete documentation on using the JEDTOPCF utility is included in the JEDTOPCF.ZIP file. The conversion process is illustrated in the following diagram:

Currently, the utility converts the information into an HP-PCF format, but this information can easily be translated into another format with some simple modifications to the output routines. A portion of a PCF file is shown below.

Table 3. Programming ispLSI devices

- 1) UBE
- 2) Execute UBE
- 3) ADDSHFT command shift
- 4) Execute ADDSHFT command
- 5) Shift address
- 6) DATASHFT command shift
- 7) Execute DATASHFT command
- 8) Shift high order data
- 9) PRGMH command shift
- 10) Execute PRGMH
- 11) DATASHFT command shift
- 12) Execute DATASHFT command
- 13) Shift low order data
- 14) PRGML command shift
- 15) Execute PRGML
- 16) Repeat from 1) until all rows are programmed.

```
unit "u1"
pcf
"x0101"
"x0111"
! Vector 100460
"x0000"
"x0010"
"x0001"
"x0011"
"x0000"
"x0010"
"x0000"
"x0010"
"x0000"
"x0010"
"x0000"
```

Contact Lattice with your tester's input vector format requirements for help in performing these modifications.

Programming ispLSI Devices with a Tester

Writing a Custom Test Program

Another approach is to develop a custom test program in the tester's language. In this example, we show how the GenRad test language can be used to program an ispLSI 1032. The GenRad language is based on PASCAL, with simple additions to control properties of the tester. To

speed up the programming time, the address shift register is not reloaded for each location. Instead, a 1 is clocked through the shift register. This saves time but requires the JEDEC file to be altered so that the first address is last. A simple AWK program, detailed in Figure 4, completes this task before the file is moved over to the tester.

Figure 4. AWK program for modifying the JEDEC file

```
#!/bin/sh
if test $# -lt 1
then
echo Usage : jedconv [filename.jed]
echo Please re-enter file name with extension !
echo
exit 1
fi
f
echo This program takes the standard Lattice JEDEC file and
echo converts it for accelerated programming. The new version
echo is saved as isp.tsr.
echo
echo converting ispLSI jedec file .....
echo
f
awk 'length($1)>79 && length($1)<81 {print $1 > "tempjed"}' $*
awk '{x[NR]=$0}
    END {for(i=NR; i>0; i=i-4)
        printf("%s\n%s\n%s\n%s\n",x[i-3],x[i-2],x[i-1],x[i])> "isp.tsr"}' tempjed
rm tempjed

echo Conversion complete.
echo
```

Programming ispLSI Devices with a Tester

The GenRad ISP Program

The following is a complete listing of a GenRad program.

(* Test and programming routine for ispLSI 1032.

Test & program sequence is :

-
1. ID-CHECK
2. FLOWTHROUGH TEST
3. BULK-ERASE
4. PROGRAM TEST S/W
5. TEST DEVICE
6. BULK-ERASE
7. PROGRAM MAIN S/W
8. VERIFY DEVICE
9. SET UES
10. SECURE

*)

test U1 dproc=d_fail_proc;

```
signal IN6, IO48, IO49, IO50, IO51, IO52, IO53, IO54, IO55,
      IO56, IO57, IO58, IO59, IO60, IO61, IO62, IO63, IN7, Y0, Y2,
      Y1, IN4, IO0, IO1, IO2, IO3, IO4, IO5, IO6, IO7, IO8, IO9,
      IO10, IO11, IO12, IO13, IO14, IO15, IN5, IO16, IO17, IO18,
      IO19, IO20, IO21, IO22, IO23, IO24, IO25, IO26, IO27, IO28,
      IO29, IO30, IO31, IO46, Y3, IO38, IO39, IO40, IO32, IO33,
      IO34, IO35, IO36, IO37, IO47, IO43, IO44, IO41, IO42, IO45,
      SDO_IN2
```

```
: hcmos_logic hcmos_currentset verify;
SDI_IN0, SCLK_IN3, MODE_IN1, ISPEN, RESETX
: hcmos_logic hcmos_currentset;
```

VAR

```
yesno      : char;
testjed    : text;
mainjed    : text;
verfout    : text;
lapse      : integer;
err_cnt    : integer;
addr_reg   : integer;
addr_num   : integer;
line_num   : integer;
char_num   : integer;
fuse_num   : integer;
veri_cnt   : integer;
data_reg   : array[1..320] of logic;
verflgic   : array[1..320] of logic;
verfchar   : array[1..320] of char;
fuse_map   : array[1..34560] of char;
```

```
cycle default interval:=500n;
  @(400n) sense()
```


Programming ispLSI Devices with a Tester

```

end;

cycle sck interval:=2.7u;
  sclk_in3 :@ (700n, 1.7u) drive (1) q0;
  sdi_in0   :@0n drive();
  mode_in1  :@0n drive();
  sdo_in2   :@2.5u sense();
  ispen     :@0n drive();
  resetx    :@0n drive();
end cycle;

cycle prog_delay interval:= 2m;
end cycle;

cycle verify_pause interval:= 30u;
end cycle;

cycle isp_sig interval:= 10u;
  ispen :@0n drive();
end cycle;

begin
  d_component:='U1';

  writeln('Initial sequence running');

  burst initialize active nomaxtime;
  begin

    (* test m1 *)

    (*****
    (* SEQUENCE 1 : ID CHECK *)
    (*****

    sck ISPEN:=1 SDI_IN0:=1 MODE_IN1:=1 RESETX:=1
      SDO_IN2=b'U; (*initialize clk*)

    isp_sig ISPEN:=0; (*ispen low for 10us to enter prog state*)

    $ ISPEN:=0 SDI_IN0:=0 MODE_IN1:=0 ; (*put device in idle state*)

    $ RESETX:=0; (*hold low throughout to prevent internal data contention*)

    sck MODE_IN1:=1; (*load device id to shift reg*)

    $ SDI_IN0:=1 MODE_IN1:=0; (*prepare to read id*)

    (*the ID for an ispLSI 1032 is 00000011. The first bit is active as soon as
    mode goes low and is the lsb. Seven more clocks will shift out the ID on
    the SDO pin, then on clk#8 the level at SDI (as it was at clk#1)
    will appear at SDO*)

    $ SDO_IN2=1; (*read 1st ID bit*)

```

Programming ispLSI Devices with a Tester

```

sck SDO_IN2=1;          (*read 2nd bit*)
sck SDO_IN2=0;          (*read 3rd bit*)
sck SDO_IN2=0;          (*read 4th bit*)
sck SDO_IN2=0;          (*read 5th bit*)
sck SDO_IN2=0;          (*read 6th bit*)
sck SDO_IN2=0;          (*read 7th bit*)
sck SDO_IN2=0;          (*read 8th bit*)
sck SDO_IN2=1;          (*SDI i/p shifted from clk#1*)
$ SDO_IN2=b'u;

(*****
(* SEQUENCE 2 : FLOWTHROUGH TEST *)
(*****)

sck MODE_IN1=1 SDI_IN0=1; (*move to shift state*)

(*load flowthru command, instruction is 01110 loading lsb first*)
sck MODE_IN1=0 SDI_IN0=0;
sck SDI_IN0=1;
sck SDI_IN0=1;
sck SDI_IN0=1;
sck SDI_IN0=0; (*load complete*)

sck MODE_IN1=1 SDI_IN0=1; (*move to execute state*)

sck MODE_IN1=0; (*execute flowthru command*)

(* check sdi = sdo *)
$ SDI_IN0=1 SDO_IN2=1;
$ SDI_IN0=0 SDO_IN2=0;
$ SDI_IN0=1 SDO_IN2=1;
$ SDI_IN0=0 SDO_IN2=0;
$ SDI_IN0=1 SDO_IN2=1;
$ SDI_IN0=0 SDO_IN2=0;
$ SDI_IN0=1 SDO_IN2=1;
$ SDI_IN0=0 SDO_IN2=0;
$ SDI_IN0=1 SDO_IN2=1;
$ SDI_IN0=0 SDO_IN2=0;
$ SDI_IN0=1 SDO_IN2=1;
$ SDI_IN0=0 SDO_IN2=0;
$ SDO_IN2=b'u;

(*****
(* SEQUENCE 3 : BULK ERASE *)
(*****)

sck MODE_IN1=1 SDI_IN0=1; (*move to shift state*)

(*load bulk erase command, instruction is 00011 loading lsb first *)
sck MODE_IN1=0 SDI_IN0=1;
sck SDI_IN0=1;
sck SDI_IN0=0;
sck SDI_IN0=0;
sck SDI_IN0=0; (*load complete*)

```

Programming ispLSI Devices with a Tester

```

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

sck MODE_IN1:=0; (*execute erase command*)

for lapse := 1 to 120 do (*wait 240ms for erase to finish*)
begin
  prog_delay;
end;

(*****)
(* SEQUENCE 4 : PROGRAM TEST S/W *)
(*****)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load address shift command, instruction is 00001 loading lsb first *)
sck MODE_IN1:=0 SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

$ MODE_IN1:=0 SDI_IN0:=0; (*execute address shift command*)

for addr_reg := 1 to 107 do (*initialize address register*)
begin
  sck;
end; (*addr reg now full of zeros*)

sck SDI_IN0:=1; (*address row 107 set to '1' and ready to program*)

sck MODE_IN1:=1 SDI_IN0:=0; (*enter idle state*)

end burst initialise; (* END OF BURST *)

writeln('Reading ISP data from file');

reset(testjed, '/work2/fk/isp/isp.data'); (*open ispdata file*)

(*load the jedec data from file to burst array*)
for line_num := 0 to 431 do (*432 lines in the ISP file*)
begin
  for char_num := 1 to 80 do (*each line is 80 chars long*)
begin
  read(testjed, fuse_map[(char_num + (80 * line_num))]); (*load the array*)
end;
readln(testjed); (*ignore carriage return at end of line*)
end;
(* the array 'fusemap' now contains the ISP file*)

(* START OF DEVICE ARRAY PROGRAMMING LOOP *)

```

Programming ispLSI Devices with a Tester

```

writeln('Programming loop running');

for addr_num := 0 to 107 do (*address loop counter*)
begin
    (*load fuse map array one address at a time to data reg array and
    simultaneously convert type 'char' to type 'logic'*)

    for fuse_num := 1 to 320 do
    begin
        if fuse_map[(fuse_num + (320 * addr_num))] = '1' then
            data_reg[fuse_num] := b'1
        else
            data_reg[fuse_num] := b'0;
        end;

    burst blow_test_function active nomaxtime inherit initialize;
    begin

    sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

    (*load data shift command, instruction is 00010 loading lsb first*)
    sck MODE_IN1:=0 SDI_IN0:=0;
    sck SDI_IN0:=1;
    sck SDI_IN0:=0;
    sck SDI_IN0:=0;
    sck SDI_IN0:=0; (*load complete*)

    sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

    $ MODE_IN1:=0 SDI_IN0:=0; (*execute data shift command*)

    (* shift in 160 high order bits for row to be programmed *)

    for fuse_num := 1 to 160 do
    begin
        sck SDI_IN0:= data_reg[fuse_num];
    end;

    sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

    (*load program high data command, instruction is 00111 loading lsb first*)
    sck MODE_IN1:=0 SDI_IN0:=1;
    sck SDI_IN0:=1;
    sck SDI_IN0:=1;
    sck SDI_IN0:=0;
    sck SDI_IN0:=0; (*load complete*)

    sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

    sck MODE_IN1:=0 SDI_IN0:=0; (*execute program high data command*)

    for lapse := 1 to 25 do (*wait 50ms for high bits of row to be programmed*)
    begin

```

Programming ispLSI Devices with a Tester

```

prog_delay;
end;

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load data shift command, instruction is 00010 loading lsb first*)
sck MODE_IN1:=0 SDI_IN0:=0;
sck SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

$ MODE_IN1:=0 SDI_IN0:=0; (*execute data shift command*)

(* shift in 160 low order bits for row to be programmed *)

for fuse_num := 161 to 320 do
begin
sck SDI_IN0:= data_reg[fuse_num];
end;

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load program low data command, instruction is 01000 loading lsb first*)
sck MODE_IN1:=0 SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=1;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*);

sck MODE_IN1:=0 SDI_IN0:=0; (*execute program low data command*)

for lapse := 1 to 25 do (*wait 50ms for low bits of row to be programmed*)
begin
prog_delay;
end;

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load address shift command, instruction is 00001 loading lsb first*)
sck MODE_IN1:=0 SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

sck MODE_IN1:=0 SDI_IN0:=0; (*move addr reg to next row, the address
reg will be clear after the last loop*)

```


Programming ispLSI Devices with a Tester

```

sck MODE_IN1:=1; (*move to idle state*)

end burst blow_test_function; (* END OF BURST *)

end; (* END OF ARRAY PROGRAMMING LOOP *)

writeln('Device programmed.');
```

(*****
 (* SEQUENCE 8 : VERIFY DEVICE *)
 (*****

```

writeln('Verifying...');

burst verification active nomaxtime inherit blow_test_function;
begin

(* device is in idle state *)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load address shift command, instruction is 00001 loading lsb first*)
sck MODE_IN1:=0 SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

$ MODE_IN1:=0; (*execute address shift command*)

sck SDI_IN0:=1; (* address last row *)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load ver/ldh command, instruction is 01010 loading lsb first*)
sck MODE_IN1:=0 SDI_IN0:=0;
sck SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=1;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

sck MODE_IN1:=0 SDI_IN0:=0; (*execute ver/ldh command*)

verify_pause; (* wait 30u for data reg to load *)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load data shift command, instruction is 00010 loading lsb first*)
sck MODE_IN1:=0 SDI_IN0:=0;

```

Programming ispLSI Devices with a Tester

```

sck SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

$ MODE_IN1:=0 SDI_IN0:=0; (*execute data shift command*)

(*clock out the high order bits from the data reg *)
for veri_cnt := 1 to 160 do
begin
$ verflgic[veri_cnt]:=sdo_in2;
sck ;
end;

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load ver/ldl command, instruction is 01011 loading lsb first*)
sck MODE_IN1:=0 SDI_IN0:=1;
sck SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=1;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

sck MODE_IN1:=0 SDI_IN0:=0; (*execute ver/ldl command*)

verify_pause; (* wait 30u for data reg to load *)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to shift state*)

(*load data shift command, instruction is 00010 loading lsb first*)
sck MODE_IN1:=0 SDI_IN0:=0;
sck SDI_IN0:=1;
sck SDI_IN0:=0;
sck SDI_IN0:=0;
sck SDI_IN0:=0; (*load complete*)

sck MODE_IN1:=1 SDI_IN0:=1; (*move to execute state*)

$ MODE_IN1:=0 SDI_IN0:=0; (*execute data shift command*)

(*clock out the low order bits from the data reg *)
for veri_cnt := 161 to 320 do
begin
$ verflgic[veri_cnt]:=sdo_in2;
sck ;
end;

sck MODE_IN1:=1 SDI_IN0:=0; (*move to idle state*)

```

Programming ispLSI Devices with a Tester

```

end burst verification; (* END OF BURST *)

(* convert data type and compare data *)

err_cnt := 0;
yesno := 'n';

for veri_cnt := 1 to 320 do
begin
    if verflgic[veri_cnt] = b'1 then (*convert type logic to type char*)
        verfchar[veri_cnt] := '1'
    else
        verfchar[veri_cnt] := '0';

    if verfchar[veri_cnt] <> fuse_map[veri_cnt] then (*compare with jedfile*)
        err_cnt := err_cnt + 1;

end;

(* failure routine *)
if err_cnt > 0 then
begin
    setfail;
    writeln('Verification failure!!!');
    writeln('failed ',err_cnt,' bit(s) out of 320.');
```

write('Write error file? [y/n]');

readln(yesno);

end

else

writeln('Verify has passed.');

(* write out error file if req'd for programmers attention*)

if yesno = 'y' then

begin

write('Writing to file...');

rewrite(verfout,'/work2/fk/isp/verify.err');

for line_num := 0 to 3 do

begin

for veri_cnt := 1 to 80 do

begin

write(verfout,verfchar[(veri_cnt + (80 * line_num))]);

end;

writeln(verfout);

end;

writeln('Done.');

writeln('Last line written to "verify.err"');

end;

(*****)

(* SEQUENCE 5 : TEST DEVICE *)

(*****)

writeln('Testing function');

Programming ispLSI Devices with a Tester

```

burst test_device active nomaxtime inherit verification;
begin
    (*test vectors here to test counter example in lattice book*)

    isp_sig ISPEN:=1; (*wait 10us to leave prog state*)

    $ SDI_IN0:=1 MODE_IN1:=1 RESETX:=1; (*hold prog pins*)

    $ Y0:=0 IO0:=1 IO1:=1 IO2:=0;
    $ IO2:=1;
    $ Y0:=1;
    $ Y0:=0; (*CNTR IS RESET O/P'S ARE LOW*)
    $ IO2:=0; (*READY TO CNT*)

    $ IO36=0 IO37=0 IO38=0 IO39=0 IO32=0;(*0000*)
    $ Y0:=1 nofails;
    $ Y0:=0 nofails;

    $ IO36=0 IO37=0 IO38=0 IO39=1 IO32=0;(*0001*)
    $ Y0:=1 nofails;
    $ Y0:=0 nofails;

    $ IO36=0 IO37=0 IO38=1 IO39=0 IO32=0;(*0010*)
    $ Y0:=1 nofails;
    $ Y0:=0 nofails;

    $ IO36=0 IO37=0 IO38=1 IO39=1 IO32=0;(*0011*)
    $ Y0:=1 nofails;
    $ Y0:=0 nofails;

    $ IO36=0 IO37=1 IO38=0 IO39=0 IO32=0;(*0100*)
    $ Y0:=1 nofails;
    $ Y0:=0 nofails;

    $ IO36=0 IO37=1 IO38=0 IO39=1 IO32=0;(*0101*)
    $ Y0:=1 nofails;
    $ Y0:=0 nofails;

    $ IO36=0 IO37=1 IO38=1 IO39=0 IO32=0;(*0110*)
    $ Y0:=1 nofails;
    $ Y0:=0 nofails;

    $ IO36=0 IO37=1 IO38=1 IO39=1 IO32=0;(*0111*)
    $ Y0:=1 nofails;
    $ Y0:=0 nofails;

    $ IO36=1 IO37=0 IO38=0 IO39=0 IO32=0;(*1000*)
    $ Y0:=1 nofails;
    $ Y0:=0 nofails;

    $ IO36=1 IO37=0 IO38=0 IO39=1 IO32=0;(*1001*)
    $ Y0:=1 nofails;

```

Programming ispLSI Devices with a Tester

```
$ Y0:=0 nofails;
$ IO36=1 IO37=0 IO38=1 IO39=0 IO32=0;(*1010*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=1 IO37=0 IO38=1 IO39=1 IO32=0;(*1011*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=1 IO37=1 IO38=0 IO39=0 IO32=0;(*1100*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=1 IO37=1 IO38=0 IO39=1 IO32=0;(*1101*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=1 IO37=1 IO38=1 IO39=0 IO32=0;(*1110*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=1 IO37=1 IO38=1 IO39=1 IO32=1;(*1111 + carry*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=0 IO37=0 IO38=0 IO39=0 IO32=0;(*0000*)
$ Y0:=1 nofails;
$ Y0:=0 nofails;

$ IO36=b'u IO37=b'u IO38=b'u IO39=b'u IO32=b'u
IO0:=b'z IO1:=b'z IO2:=b'z Y0:=b'z
SDI_IN0:=b'z MODE_IN1:=b'z RESETX:=b'z ISPEN:=b'z;

end burst test_device; (* END OF BURST *)

writeln('Finished');

end test U1;
```


Section 1: Introduction

Section 2: ispLSI and pLSI Architecture Overview

Section 3: ispLSI and pLSI Development Tools

Section 4: ispLSI and pLSI Application Notes

Section 5: GAL Architecture Overview

Introduction to Generic Array Logic 5-1

Section 6: GAL Development Tools

Section 7: GAL Application Notes

Section 8: In-System Programmable Generic Digital Switch (ispGDS)

Section 9: Design Techniques

Section 10: Article Reprints

Section 11: Technology, Quality, and Reliability Overview

Section 12: General Section

Section 1: General Section	
Section 1.1: Technology, Quality, and Reliability Overview	
Section 1.2: Article Reprints	
Section 2: Design Techniques	
Section 3: In-System Programmable Generic Digital Switch (ispGDS)	
Section 4: GAL Application Notes	
Section 5: GAL Development Tools	
Section 6: Introduction to Generic Array Logic	5-1
Section 7: GAL Architecture Overview	
Section 8: ispLSI and ispLSI Application Notes	
Section 9: ispLSI and ispLSI Development Tools	
Section 10: ispLSI and ispLSI Architecture Overview	
Section 11: Introduction	

Introduction to Generic Array Logic

Overview

In 1985, Lattice introduced a new type of programmable logic device (PLD) that transformed the PLD market: the Generic Array Logic (GAL) device. The E²CMOS technology of the GAL devices gave them significant advantages over their bipolar PAL counterparts; not only could GAL devices be programmed quickly and efficiently, but they could also be erased and reprogrammed. Today, Lattice is the leading supplier, worldwide, of low-density PLDs. Industry leading performance, low power E²CMOS technology, 100% testability and 100% programming yields make the GAL family the preferred choice among system designers.

The GAL family includes fourteen distinct product architectures, with a variety of performance levels specified across commercial, industrial, and military (MIL-STD-883) operating ranges, to meet the demands of any system logic design.

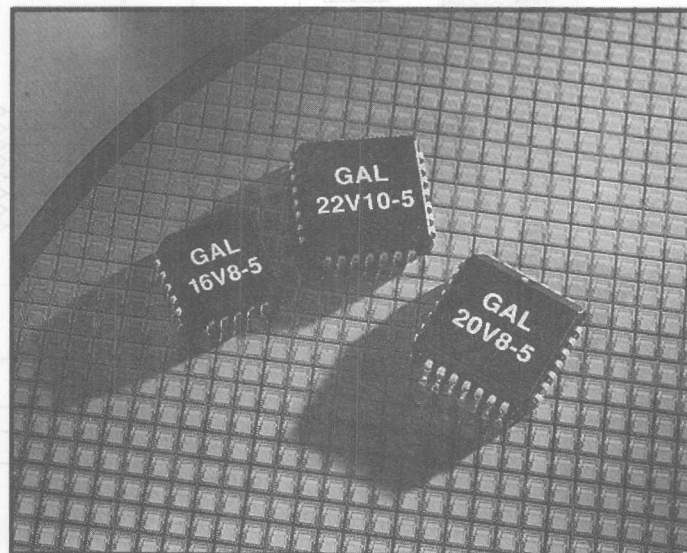
These GAL products can be segmented into two broad categories:

Base Products - Aimed at providing superior design alternatives to bipolar PLDs, these five architectures replace over 98% of all bipolar PAL devices. The GAL16V8 and GAL20V8 replace forty-two different PAL devices. The GAL22V10, GAL20RA10, and GAL20XV10 round out the base products. These GAL devices meet and, in most cases, beat bipolar PAL performance specifications while consuming significantly lower power and offering higher quality and reliability via Lattice's electrically reprogrammable E²CMOS technology. High speed erase times (<100ms) allow the devices to be reprogrammed quickly and efficiently.

Extension Products - These products build upon the Base GAL product features to provide enhanced functionality including innovative architectures (GAL18V10, GAL26CV12, GAL6001/6002), 64mA high output drive (GAL16VP8 & GAL20VP8), "Zero power" operation (GAL16V8Z/ZD & GAL20V8Z/ZD) and in-system programmability (ispGAL22V10).

A Product for any System Design Need

Lattice GAL products have the performance, architectural features, low power, and high quality to meet the needs of the most demanding system designs.



Lattice offers the broadest line of high-performance PLDs.

Introduction to Generic Array Logic

The GAL16V8 and GAL20V8

The GAL16V8 (20-pin) and GAL20V8 (24-pin) provide the highest speed performance available in the PLD market. CMOS circuitry allows the GAL16V8 and GAL20V8 low power devices to consume just 75mA typical Icc, which represents a 50% savings in power when compared to bipolar counterparts. Quarter power versions save even more at 45mA Icc.

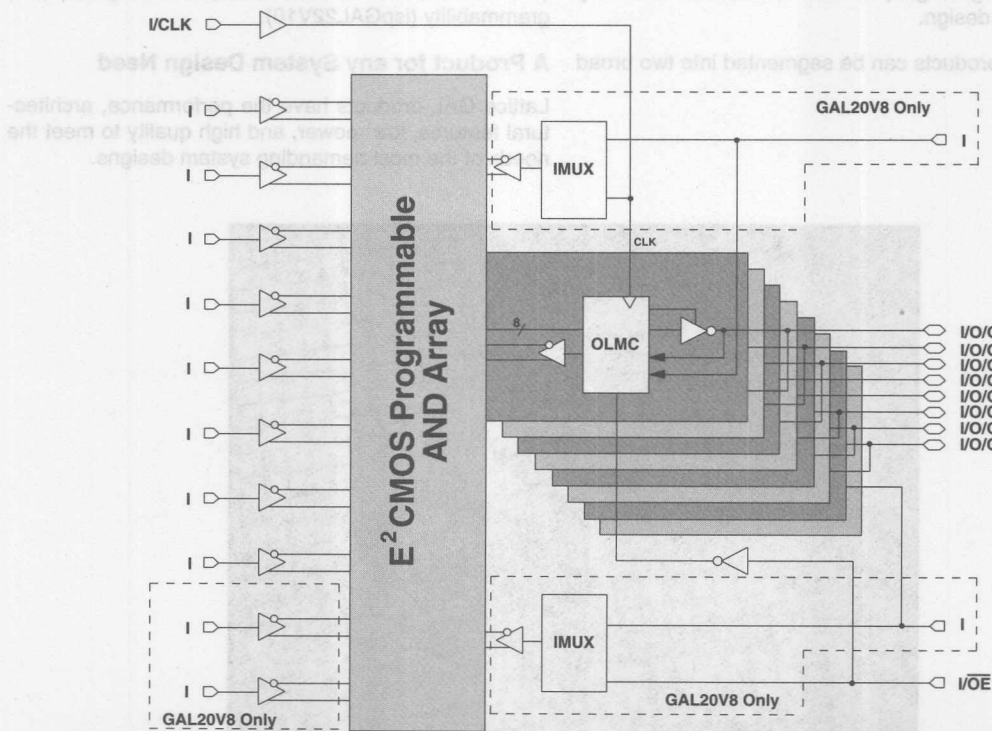
The GAL16V8 is a 20-pin device which contains eight dedicated input pins and eight I/O pins. The GAL20V8 is a 24-pin version of the 16V8 device with twelve dedicated input pins and eight I/O pins. Their generic architecture provides maximum design flexibility by allowing the Output Logic Macrocell (OLMC) to be configured by the user. An important subset of the many architecture configurations possible with the GAL16V8 and GAL20V8 are the standard PAL architectures. Providing eight OLMCs with eight product terms each, GAL16V8 and GAL20V8 de-

vices are capable of emulating virtually all PAL architectures with full function/fuse map/parametric compatibility.

Output Logic Macrocell

There are three OLMC configuration modes possible in GAL16V8 and GAL20V8 devices: registered, complex, and simple. These are illustrated in the diagrams on the following pages. You cannot mix modes; all OLMCs are either simple, complex, or registered (in registered mode, the output can be combinational or registered).

The outputs of the AND array are fed into an OLMC, where each output can be individually set to active high or active low, with either combinational (asynchronous) or registered (synchronous) configurations. A common output enable is connected to all registered outputs, or a product term can be used to provide individual output



GAL16V8 and GAL20V8 Block Diagram

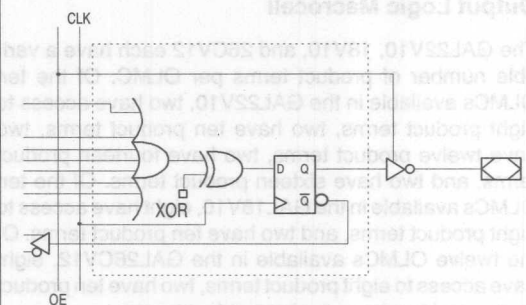
Introduction to Generic Array Logic

enable control for combinational outputs in the registered mode or combinational outputs in the complex mode.

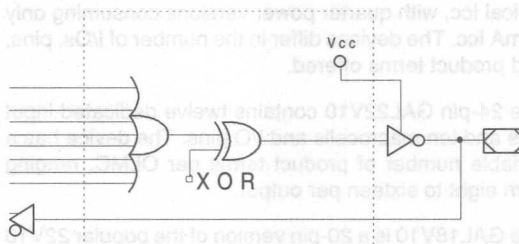
There is no output enable control in the simple mode. The OLMC provides the designer with maximum output flex-

ibility in matching signal requirements, thus providing more functionality than possible with standard PAL devices.

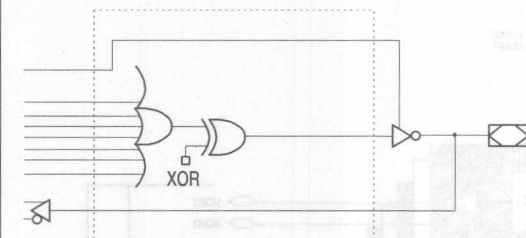
Registered Configuration for Registered Mode



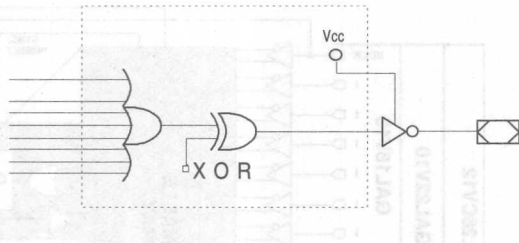
Combinatorial Output with Feedback Configuration for Simple Mode



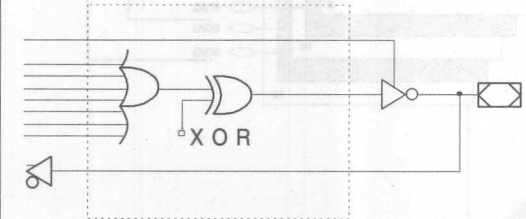
Combinatorial Configuration for Registered Mode



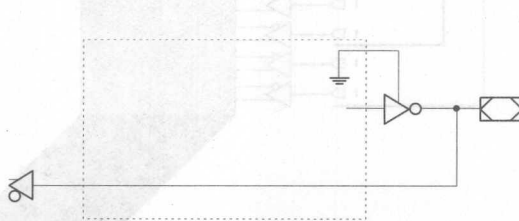
Combinatorial Output Configuration for Simple Mode



Combinatorial Output Configuration for Complex Mode



Dedicated Input Configuration for Simple Mode



Introduction to Generic Array Logic

The GAL22V10, GAL18V10 and GAL26CV12

Three devices are offered in the high-speed, E²CMOS GAL22V10 family: the GAL22V10 (24-pin), GAL18V10 (20-pin), and GAL26CV12 (28-pin). Each of these devices uses the industry standard 22V10 universal architecture, which provides maximum design flexibility by allowing the OLMC to be configured by the user. The GAL22V10 family low power devices consume just 90mA typical I_{cc}, with quarter power versions consuming only 45mA I_{cc}. The devices differ in the number of I/Os, pins, and product terms offered.

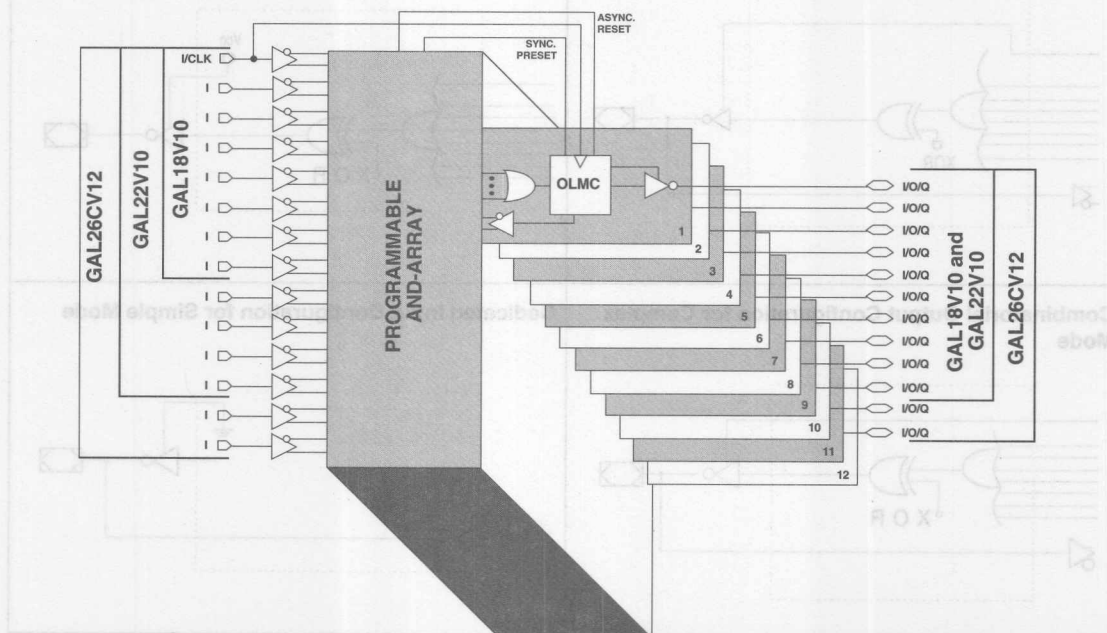
The 24-pin GAL22V10 contains twelve dedicated input pins and ten macrocells and I/O pins. The device has a variable number of product terms per OLMC, ranging from eight to sixteen per output.

The GAL18V10 is a 20-pin version of the popular 22V10 device. It provides a smaller footprint and lower cost alternative to the 22V10 device. The GAL18V10 contains eight dedicated input pins and ten macrocells and I/O pins.

The GAL26CV12 is a 28-pin version of the 22V10 device. It features more inputs and outputs in order to provide greater functionality and increased I/O. The GAL26CV12 contains fourteen dedicated input pins and twelve macrocells and I/O pins.

Output Logic Macrocell

The GAL22V10, 18V10, and 26CV12 each have a variable number of product terms per OLMC. Of the ten OLMCs available in the GAL22V10, two have access to eight product terms, two have ten product terms, two have twelve product terms, and two have sixteen product terms. Of the ten OLMCs available in the GAL18V10, eight have access to eight product terms, and two have ten product terms. Of the twelve OLMCs available in the GAL26CV12, eight have access to eight product terms, two have ten product terms, and two have twelve product terms.



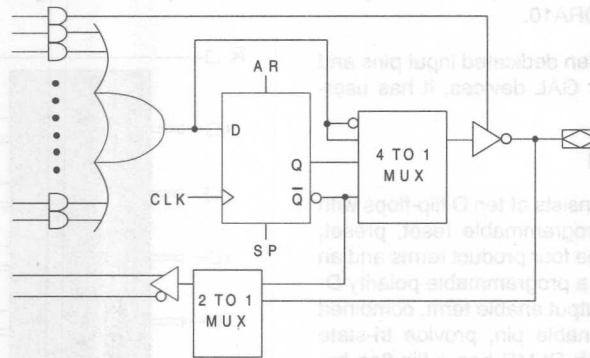
GAL22V10, GAL18V10 and GAL26CV12 Block Diagram

Introduction to Generic Array Logic

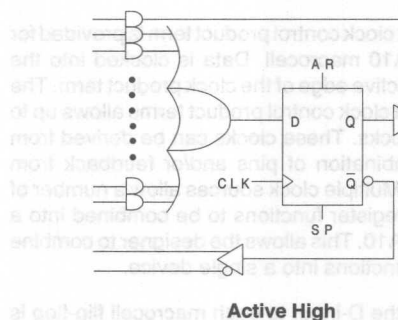
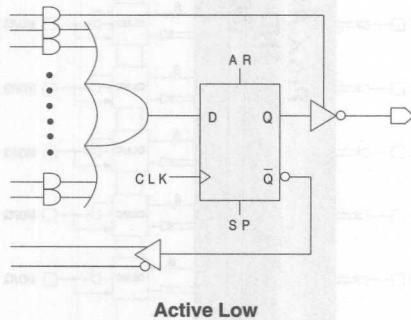
The output polarity of each OLMC can be individually programmed to be true or inverting, in either combinational or registered mode. This allows the user to reduce the overall number of product terms required in a design and/or to invert the output signal.

GAL22V10 family devices have a product term for Asynchronous Reset (AR) and a product term for Synchronous Preset (SP). These two product terms are common to all registered OLMCs.

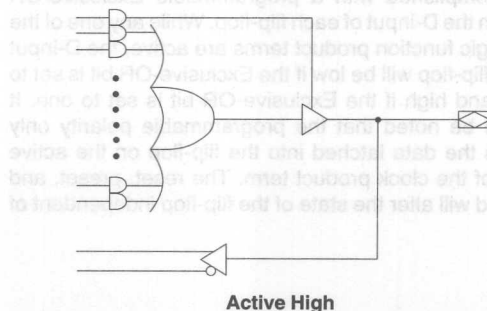
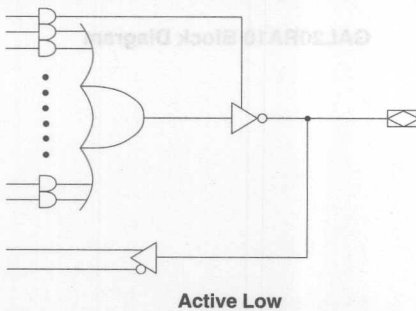
GAL22V10, GAL18V10 and GAL26CV12 Output Logic Macrocell



Output Logic Macrocell Configuration (Registered Mode)



Output Logic Macrocell Configuration (Combinatorial Mode)



Introduction to Generic Array Logic

The GAL20RA10

The GAL20RA10 (24-pin) supports high performance, asynchronous logic. It is a direct parametric compatible CMOS replacement for the PAL20RA10 device. However, Lattice's E²CMOS circuitry achieves power levels as low as 75mA typical I_{cc}, which represents a substantial savings in power when compared to bipolar counterparts like the PAL20RA10.

The GAL20RA10 contains ten dedicated input pins and ten I/O pins. As with other GAL devices, it has user-configurable OLMCs.

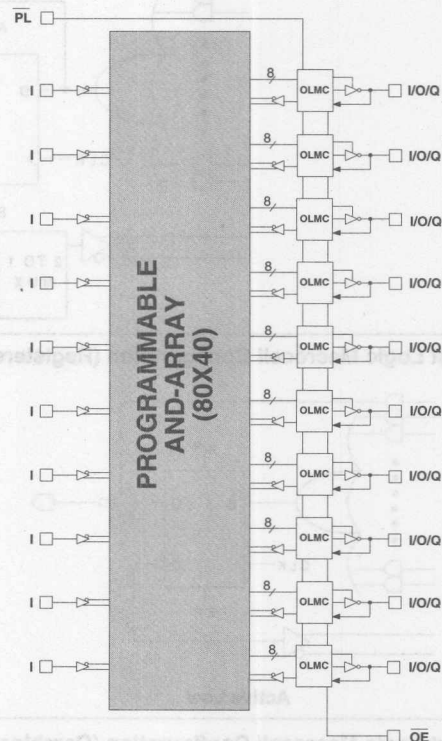
Output Logic Macrocell

The GAL20RA10 OLMC consists of ten D flip-flops with individual asynchronous programmable reset, preset, and clock product terms. The four product terms and an Exclusive-OR gate provide a programmable polarity D-input to each flip-flop. An output enable term, combined with a dedicated output enable pin, provide tri-state control of each output. Each OLMC has a flip-flop bypass, allowing any combination of registered or combinational outputs.

An independent clock control product term is provided for each GAL20RA10 macrocell. Data is clocked into the flip-flop on the active edge of the clock product term. The use of individual clock control product terms allows up to ten separate clocks. These clocks can be derived from any pin or combination of pins and/or feedback from other flip-flops. Multiple clock sources allow a number of asynchronous register functions to be combined into a single GAL20RA10. This allows the designer to combine discrete logic functions into a single device.

The polarity of the D-input to each macrocell flip-flop is individually programmable to be active high or low. This is accomplished with a programmable Exclusive-OR gate on the D-input of each flip-flop. While any one of the four logic function product terms are active, the D-input to the flip-flop will be low if the Exclusive-OR bit is set to zero, and high if the Exclusive-OR bit is set to one. It should be noted that the programmable polarity only affects the data latched into the flip-flop on the active edge of the clock product term. The reset, preset, and preload will alter the state of the flip-flop independent of

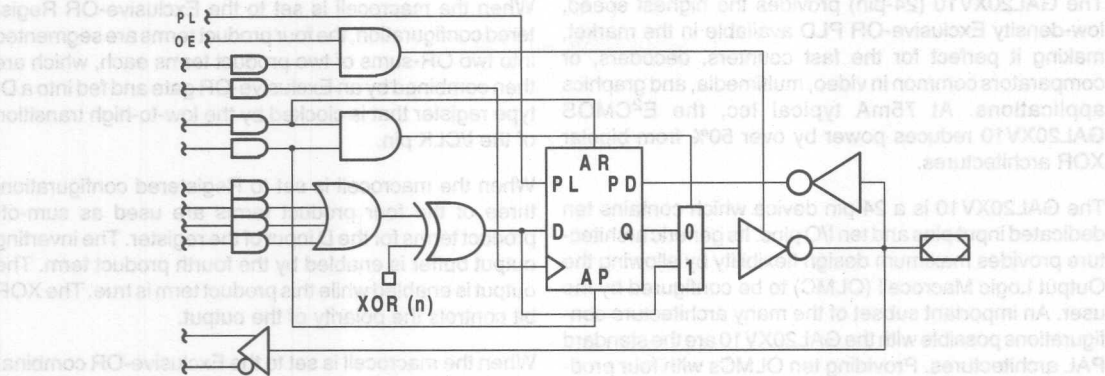
the state of the programmable polarity bit. The ability to program the active polarity of the D-inputs can be used to reduce the total number of product terms used, by allowing the DeMorganization of the logic functions. This logic reduction is accomplished by the logic compiler, and does not require the designer to define the polarity.



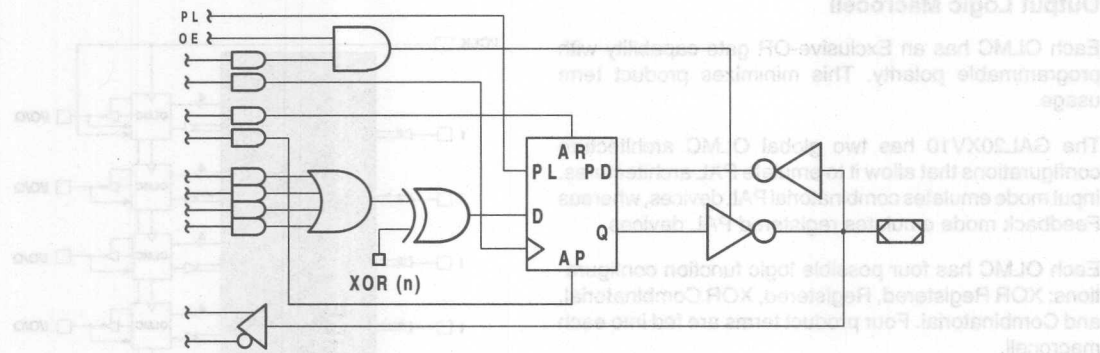
GAL20RA10 Block Diagram

Introduction to Generic Array Logic

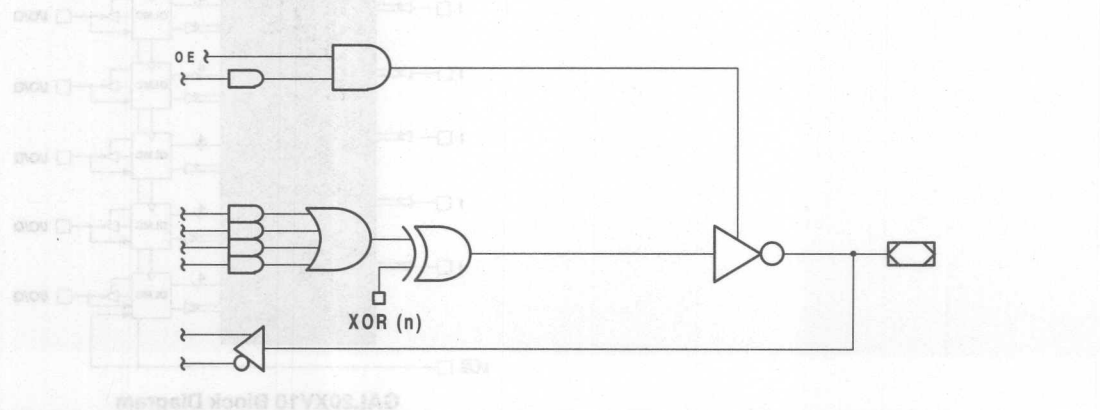
GAL20RA10 Output Logic Macrocell Diagram



Output Logic Macrocell Configuration (Registered with Polarity)



Output Logic Macrocell Configuration (Combinatorial with Polarity)



Introduction to Generic Array Logic

The GAL20XV10

The GAL20XV10 (24-pin) provides the highest speed, low-density Exclusive-OR PLD available in the market, making it perfect for the fast counters, decoders, or comparators common in video, multimedia, and graphics applications. At 75mA typical I_{CC} , the E^2 CMOS GAL20XV10 reduces power by over 50% from bipolar XOR architectures.

The GAL20XV10 is a 24-pin device which contains ten dedicated input pins and ten I/O pins. Its generic architecture provides maximum design flexibility by allowing the Output Logic Macrocell (OLMC) to be configured by the user. An important subset of the many architecture configurations possible with the GAL20XV10 are the standard PAL architectures. Providing ten OLMCs with four product terms each, the GAL20XV10 is capable of emulating the PAL12L10, PAL20L10, PAL20X10, PAL20X8, and PAL20X4 devices.

Output Logic Macrocell

Each OLMC has an Exclusive-OR gate capability with programmable polarity. This minimizes product term usage.

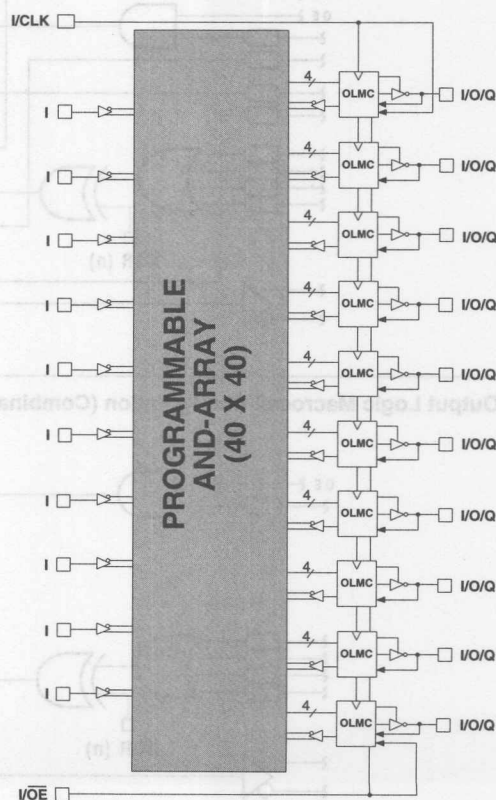
The GAL20XV10 has two global OLMC architecture configurations that allow it to emulate PAL architectures. Input mode emulates combinatorial PAL devices, whereas Feedback mode emulates registered PAL devices.

Each OLMC has four possible logic function configurations: XOR Registered, Registered, XOR Combinatorial, and Combinatorial. Four product terms are fed into each macrocell.

When the macrocell is set to the Exclusive-OR Registered configuration, the four product terms are segmented into two OR-sums of two product terms each, which are then combined by an Exclusive-OR gate and fed into a D-type register that is clocked by the low-to-high transition of the I/CLK pin.

When the macrocell is set to Registered configuration, three of the four product terms are used as sum-of-product terms for the D input of the register. The inverting output buffer is enabled by the fourth product term. The output is enabled while this product term is true. The XOR bit controls the polarity of the output.

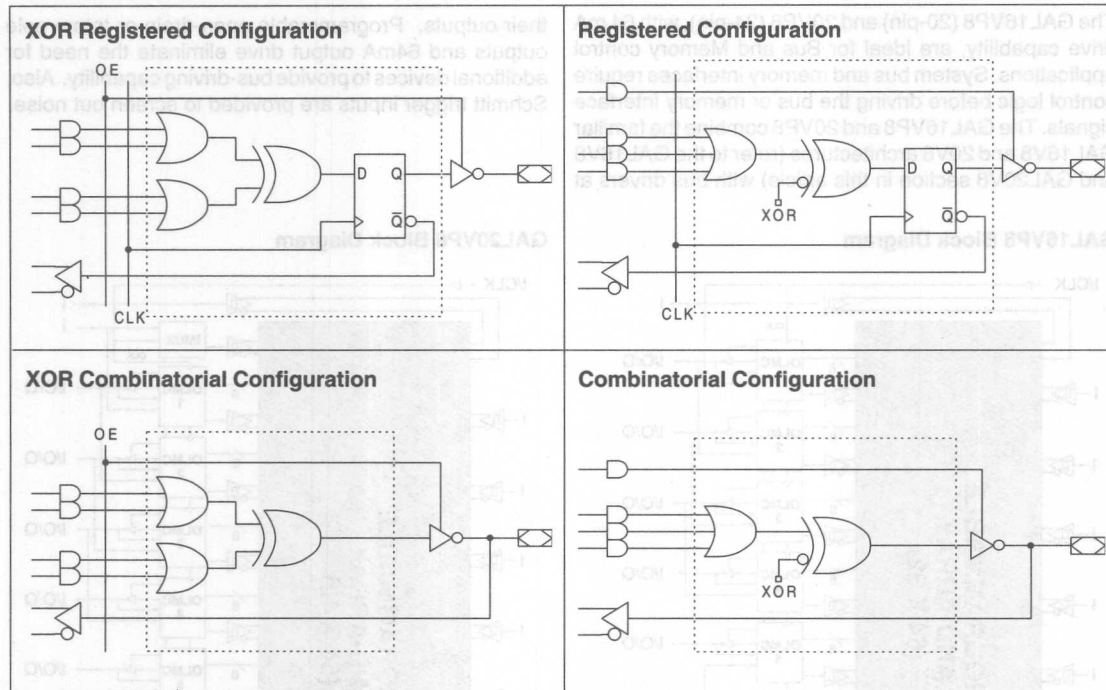
When the macrocell is set to the Exclusive-OR combinatorial configuration, the four product terms are segmented into two OR-sums of two product terms each, which are then combined by an Exclusive-OR gate and fed to an output buffer.



GAL20XV10 Block Diagram

Introduction to Generic Array Logic

GAL20XV10 OLMC Configurations



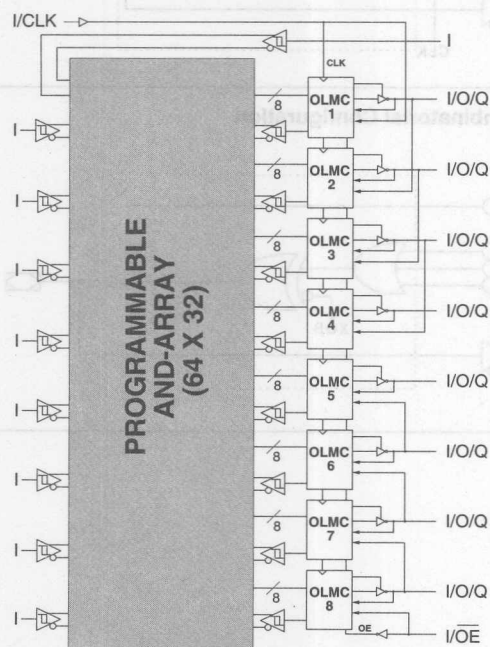
Introduction to Generic Array Logic

The GAL16VP8 and GAL20VP8

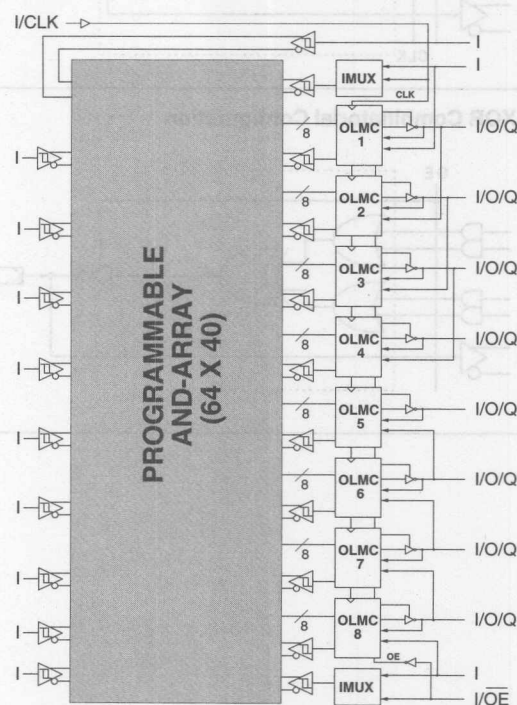
The GAL16VP8 (20-pin) and 20VP8 (24-pin), with 64 mA drive capability, are ideal for Bus and Memory control applications. System bus and memory interfaces require control logic before driving the bus or memory interface signals. The GAL16VP8 and 20VP8 combine the familiar GAL16V8 and 20V8 architectures (refer to the GAL16V8 and GAL20V8 section in this article) with bus drivers at

their outputs. Programmable open-drain or totem pole outputs and 64mA output drive eliminate the need for additional devices to provide bus-driving capability. Also, Schmitt trigger inputs are provided to screen out noise.

GAL16VP8 Block Diagram



GAL20VP8 Block Diagram



Introduction to Generic Array Logic

The GAL16V8Z/ZD and GAL20V8Z/ZD

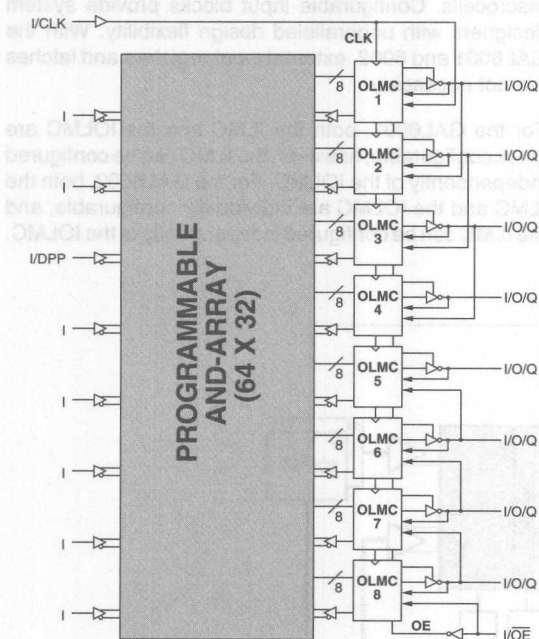
The GAL16V8Z/ZD (20-pin) and GAL20V8Z/ZD (24-pin), at 100uA standby current, provide the highest speed and lowest power combination PLDs available in the market. These devices are ideal for battery powered systems.

The GAL16V8Z and 20V8Z use Input Transition Detection (ITD) to put the device in standby mode and are capable of emulating the full functionality of the standard

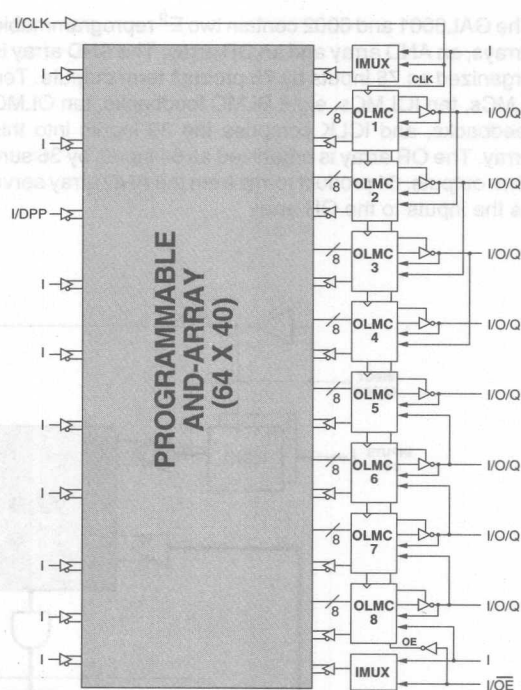
GAL16V8 and 20V8 respectively (refer to the GAL16V8 and GAL20V8 section in this article). The GAL16V8ZD and 20V8ZD utilize a dedicated power-down pin (DPP) to put the device in standby mode.

The GAL16V8ZD has 15 inputs available to the AND array, whereas the GAL20V8ZD has 19 inputs available to the AND array.

GAL16V8Z/ZD Block Diagram



GAL20V8Z/ZD Block Diagram



Introduction to Generic Array Logic

The GAL6001 and GAL6002

Having an FPLA architecture, known for its superior flexibility in state-machine design, the GAL6001 (24-pin) and GAL6002 (24-pin) offer a high degree of functional integration and flexibility in a 24-pin device.

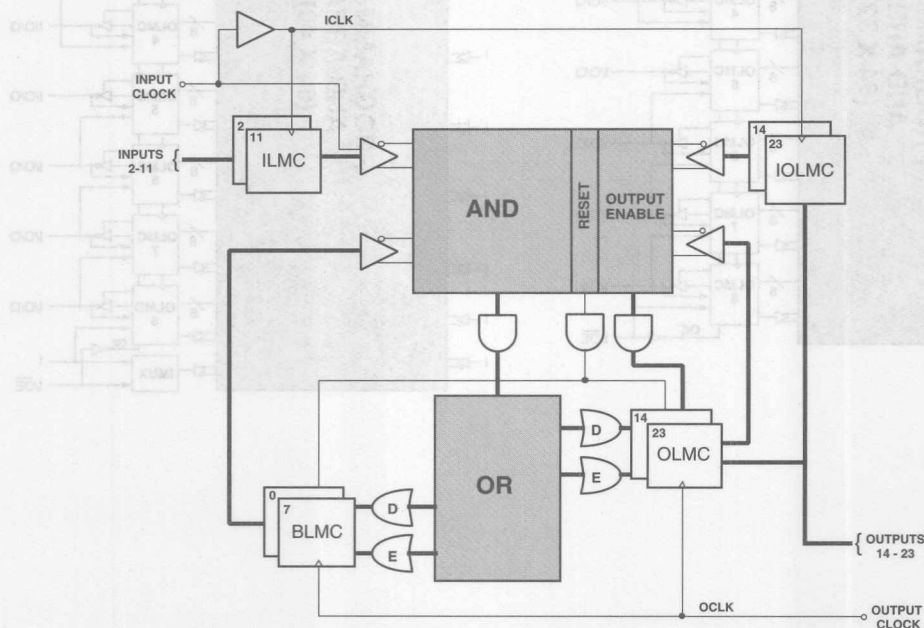
The GAL6001 and GAL6002 have ten programmable Output Logic Macrocells (OLMCs) and eight programmable Buried Logic Macrocells (BLMCs). In addition, there are ten Input Logic Macrocells (ILMCs) and ten I/O Logic Macrocells (IOLMCs). Two clock inputs are provided for independent control of the input and output macrocells.

The GAL6001 and 6002 contain two E^2 reprogrammable arrays, an AND array and an OR array. The AND array is organized as 78 inputs by 75 product term outputs. Ten ILMCs, ten IOLMCs, eight BLMC feedbacks, ten OLMC feedbacks, and ICLK comprise the 39 inputs into this array. The OR array is organized as 64 inputs by 36 sum term outputs. 64 product terms from the AND array serve as the inputs to the OR array.

Input Logic Macrocell (ILMC) and I/O Logic Macrocell (IOLMC)

The GAL6001 and 6002 feature two configurable input sections. The ILMC section corresponds to the dedicated input pins, and the IOLMC section corresponds to the I/O pins. On the GAL6001, each input section is configurable as a block for asynchronous, latched, or registered inputs. On the GAL6002, however, each input section is individually configurable as asynchronous, latched, or registered inputs. ICLK is used as an enable input for latched macrocells or as a clock input for registered macrocells. Configurable input blocks provide system designers with unparalleled design flexibility. With the GAL6001 and 6002, external input registers and latches are not necessary.

For the GAL6001, both the ILMC and the IOLMC are block configurable; however, the ILMC can be configured independently of the IOLMC. For the GAL6002, both the ILMC and the IOLMC are individually configurable, and the ILMC can be configured independently of the IOLMC.



GAL6001 and GAL6002 Block Diagram

Introduction to Generic Array Logic

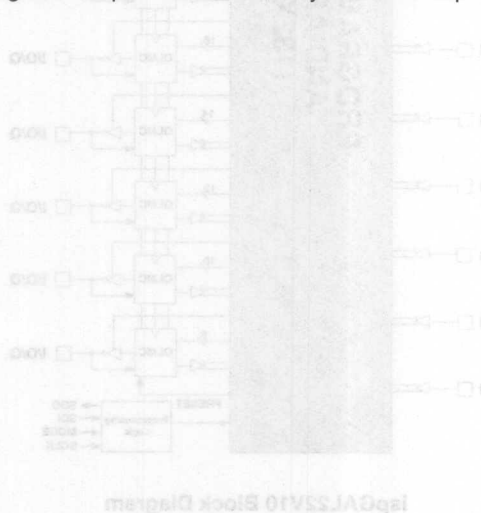
Output Logic Macrocell (OLMC) and Buried Logic Macrocell (BLMC)

The outputs of the OR array feed two groups of macrocells. One group of eight macrocells is buried; its outputs feed back directly into the AND array rather than to device pins. These cells are called the Buried Logic Macrocells (BLMCs), and are useful for building state machines. The second group of macrocells consists of ten cells whose outputs, in addition to feeding back into the AND array, are available at the device pins. Cells in this group are known as Output Logic Macrocells (OLMCs).

The Output and Buried Logic Macrocells are configurable on a macrocell by macrocell basis. They may be set to one of three configurations: combinatorial, D-type register with sum term (asynchronous) clock, or D/E-type register. Output macrocells always have I/O capability,

with directional control provided by the ten output enable (OE) product terms. Additionally, the polarity of each OLMC output is selected through the "D" XOR. Polarity selection is available for BLMCs, since both the true and complemented forms of their outputs are available in the AND array. Polarity of all "E" sum terms is selected through the "E" XOR.

Registers in both the OLMCs and BLMCs feature a common RESET product term. This active high product term allows the registers to be asynchronously reset. Registers are reset to a logic zero. If connected to an output pin, a logic one will occur because of the inverting output buffer.

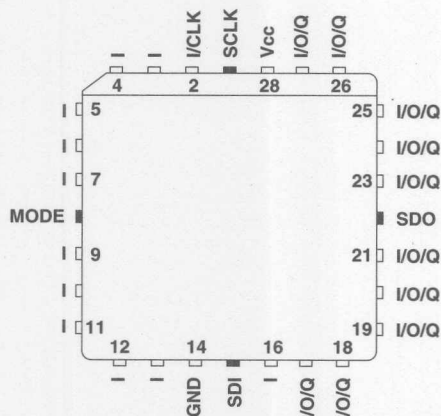


Introduction to Generic Array Logic

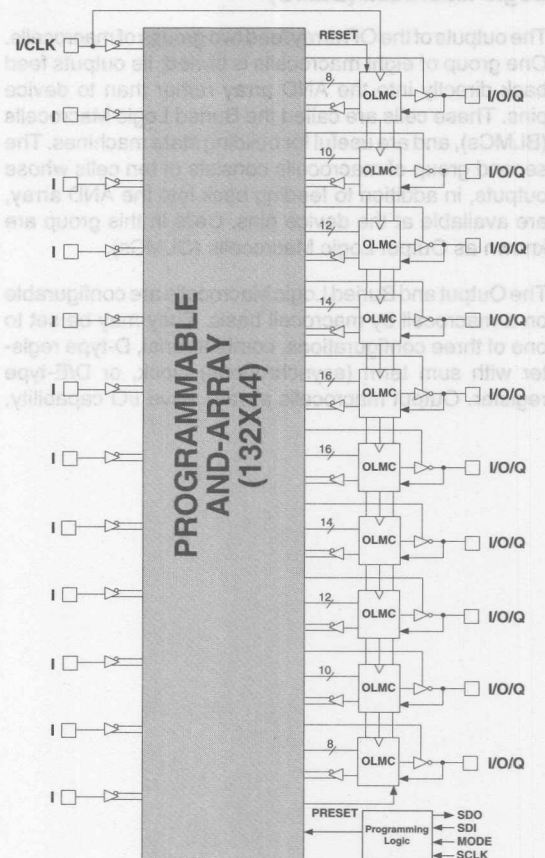
The ispGAL22V10

The ispGAL22V10 (28-pin) provides the industry's first in-system programmable 22V10 device. It is fully function/fuse map/parametric compatible with standard bipolar and CMOS 22V10 devices (refer to the GAL22V10, GAL18V10, and GAL26CV12 section in this article). The standard 28-pin PLCC package provides the same functional pinout at the standard 22V10 PLCC package with the four No-Connect pins being used for ISP interface signals.

The in-system programming capability of the ispGAL22V10 allows designers to define and develop systems with capabilities previously unattainable. ISP provides the ability to program and reprogram logic devices while attached to the printed circuit board (PCB). No other logic technology is better for reducing time to market, while assuring the highest system quality and lowest overall cost. With ISP technology, hardware as flexible and easy to modify as software becomes a reality: hardware functions can be programmed and modified in real time to expand product features, shorten system design and debug time, enhance product manufacturability and simplify field upgrades.



ispGAL22V10 Pinout Diagram



ispGAL22V10 Block Diagram

Section 1: Introduction

Section 2: ispLSI and pLSI Architecture Overview

Section 3: ispLSI and pLSI Development Tools

Section 4: ispLSI and pLSI Application Notes

Section 5: GAL Architecture Overview

Section 6: GAL Development Tools

 Using GAL Development Tools 6-1

 GAL Development Support 6-11

 Copying PAL, EPLD and PEEL Patterns into GAL Devices 6-13

Section 7: GAL Application Notes

Section 8: In-System Programmable Generic Digital Switch (ispGDS)

Section 9: Design Techniques

Section 10: Article Reprints

Section 11: Technology, Quality, and Reliability Overview

Section 12: General Section

Section 1: Introduction	
Section 2: iapLSI and pLSI Architecture Overview	
Section 3: iapLSI and pLSI Development Tools	
Section 4: iapLSI and pLSI Application Notes	
Section 5: GAL Architecture Overview	
Section 6: GAL Development Tools	
Using GAL Development Tools	6-1
GAL Development Support	6-11
Copying PAL, EPROM and PEEP Patterns into GAL Devices	6-13
Section 7: GAL Application Notes	
Section 8: In-System Programmable Generic Digital Switch (ispGDS)	
Section 9: Design Techniques	
Section 10: Article Reprints	
Section 11: Technology, Quality, and Reliability Overview	
Section 12: General Section	

Using GAL Development Tools

GAL Hardware and Software Tools

Lattice Semiconductor specializes in the design and manufacture of high-speed E²CMOS programmable logic devices. It is not our intention to provide custom software and/or hardware for the purpose of developing patterns for the GAL family of devices, and, as such, we leave the software/hardware task to the respective third-party experts in those fields.

Such universal tool suppliers provide support for all major devices, from a variety of manufacturers. If you're just starting out with programmable logic, and plan to purchase development tools, rest assured that industry-standard hardware and software will handle GAL device development. If you're already using standard tools for PLD development, the move to GAL devices won't require sophisticated or expensive upgrades; current third-party development tools support Lattice GAL devices to their full extent. At most, an upgrade to the current revision of the support tool may be required.

Lattice's Applications Department remains on call to assist you in the task of logic development using third-party tools. Our engineers, trained on a variety of standard equipment, are prepared to answer any questions you may have. In addition, they are able to use the tools to more fully exploit the unique benefits of GAL devices. Here we provide the basis for getting started with GAL

devices. As you proceed with the development of your applications, call us - we'd like to hear how it's going.

The typical PLD design flow, shown in figure 1, begins with a design specification, iterates the logic to achieve proper functionality, and ends with a 'download' of the information to a programming fixture that patterns the device for the system. Critical to the accuracy and ultimate success of the PLD design process is the use of logic development tools to minimize the chance of error and improve design efficiency.

Software Tools

In the early days of programmable logic devices, fuse maps were entered by hand on a piece of paper with a fuse map table, and then manually transferred into the hardware programmers. This basic concept of fuse map generation is still valid for modern devices, such as Lattice's GAL devices or any other PLDs, but the software tools have greatly advanced from those early days. Although the final result of any software tool is the generation of a fuse map, there are many methods in which logic designs can be entered. Most third party logic compiler software environments offer Boolean equation, truth table and state machine design entry methods. These three basic design entry methods vastly improved the efficiency and accuracy of logic design. To further improve efficiency of design entry, newer software packages offer schematic entry, macro library, timing waveform, and hardware description language (HDL) design entry methods. By combining these multiple design entry methods within each third party software package, system design engineers are able to select the method that suits his or her logic design. Accuracy of the logic implementation is further improved by the ability to perform functional and timing simulation within the software.

Software packages, such as ABEL from Data I/O, CUPL from Logical Devices, PLDesigner from Minc, and OrCAD-PLD from OrCAD, provide various combinations of the above mentioned design entry methods. With these software packages, the required logic design can be fully designed, simulated, and debugged using software, before any hardware is built. All of these software packages perform a similar function; they process and synthesize the design idea entered by the specified method, convert this result into an intermediate file, such as netlist or PLA file, and finally generate the fuse map file for the programmer. As part of this process, a documentation file is also created. As we are moving in to the future with higher

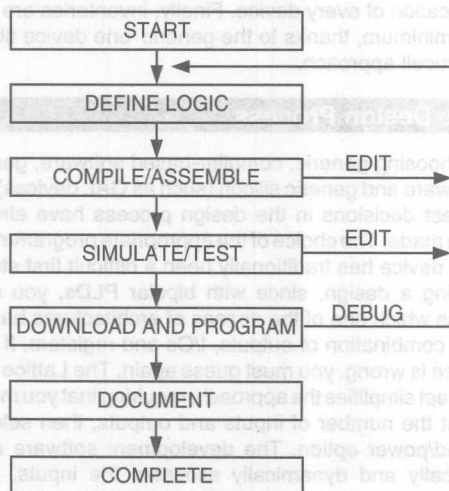


Figure 1. PLD Design Flow

Using GAL Development Tools

density PLDs, third party supplied universal software tools are becoming more of an integral part of the hardware.

Hardware Tools

The same arguments as those expressed above for universal software tools apply to universal hardware. Hardware that is developed by third parties is more flexible and provides a future growth path to the user. Lattice recommends the use of third party programming hardware for GAL device development.

Universal programming hardware allows the programming of a variety of devices without the aid of custom fixtures or manufacturers' adapters. Since the Lattice GAL programming algorithm requires no abnormal voltages or timings, as some one-time programmable technologies do, most all hardware manufacturers support GAL devices on existing models.

Patterning the PLD is the process of providing it with the data (the JEDEC file) to perform a specific custom function, and applying the appropriate series of voltage pulses. 'Support' by the hardware manufacturer refers to his ability to provide the appropriate voltage pulses and timings for a given PLD. After that, patterning a device merely requires downloading the JEDEC file.

Downloading is the process of 'teaching' the hardware programmer the pattern that is necessary to program a device. This data can come from a pre-patterned device (or 'master'), from a computer via direct connection or modem, or from an attached peripheral, such as a floppy disk. If the file is transferred in JEDEC format, as most are, a checksum is calculated and verified at the end of the data transfer to ensure that no data was dropped or garbled during transmission. Most programmers have either a single button or simple command string that puts the hardware into the download mode.

The programming of the GAL device is controlled by the programming hardware. Since the GAL device uses a nonvolatile, reprogrammable E²CMOS technology, the device can be erased; in fact, the device is automatically erased as the first step in the programming algorithm.

The patterning of the GAL device array is done using a parallel-programming scheme, which keeps the total programming time to well under a second. The algorithm is so efficient that it programs devices nearly 50% faster than typical bipolar PLD algorithms, and an order or magnitude faster than UV-CMOS approaches. During this programming time, both the logic array and the architecture matrix are patterned.

Finally, an analog verify of each and every cell in the GAL device takes place, to ensure that the cell is fully programmed and will retain data for a minimum of 20 years.

It is worth noting here that GAL devices offer a security cell that can be programmed to prevent examination (or further verification) of the pattern in the programmable arrays — a feature provided so that a proprietary design can be obscured from competitive or enemy eyes.

Somewhat ironically, the GAL device security cell is itself erasable; it can only be erased, however, in conjunction with an array 'Bulk Erase,' during which all bits are cleared at once. This allows the designer or manufacturing person to reuse previously secured devices — a feature never before available in PLDs.

Debugging and Pattern Revisions

GAL devices bring extensive advantages to the manufacturing and design engineering areas, due to their unique combination of E²CMOS technology, generic architecture, and unmatched quality levels. Only GAL devices are instantly erasable in a standard hardware programming fixture. As mentioned, erasure takes place automatically just prior to the re-patterning of the array. No time-consuming 'trips to a UV lamp' are necessary, as with UV-erasable PLDs. Both the GAL device's logic array and device architecture configuration are fully reprogrammable and reconfigurable. In addition, the erasable GAL device is assembled in a low-cost plastic package, not an expensive quartz-windowed package. Pattern revisions can be recorded in the device's electronic signature, allowing the traceability, tracking and verification of every device. Finally, inventories are kept to a minimum, thanks to the generic 'one device fits all' macrocell approach.

The Design Process

By choosing generic, compiler-based software, generic hardware and generic silicon (such as GAL devices), the biggest decisions in the design process have already been made. The choice of the appropriate programmable logic device has traditionally been a difficult first step in starting a design, since with bipolar PLDs, you must guess which one of the dozens of architectures has the right combination of outputs, I/Os and registers. If your choice is wrong, you must guess again. The Lattice GAL concept simplifies the approach, requiring that you merely count the number of inputs and outputs, then select a speed/power option. The development software automatically and dynamically allocates the inputs, I/Os, registers, and so on.

Using GAL Development Tools

The following design example shows how to implement two basic logic gates in a GAL device. The specific syntax is that of ABEL; however, other generic software (CUPL) has similar syntax and functions. In this cursory 'walk-through', segments of code are presented as they would appear on the screen of the personal computer running the ABEL software. The manufacturers of the software would, of course, be glad to provide a more comprehensive tutorial.

All the design processes are invoked from the ABEL design environment as shown in figure 2. The File Menu allows the user to manipulate file options such as opening, merging, and closing design files.

Once you open a new design file, fields are normally provided for optional information, such as company name, design description, etc.:

```
module gates
Title 'Tutorial Using a GAL16V8
      Lattice Semiconductor'
```

The device, its pinout, pin labels, and intermediate variables need to be specified next. Use names that are convenient for you to reference, since the software doesn't care what you call a pin, as long as you are consistent:

```

gates device 'p16v8';

"**** inputs ****

    A, !B PIN 1,2;

"**** outputs ****

    Y, !Z PIN 18,19 ISTYPE
'COM,INVERT';

"**** intermediate definitions ****

    C,X,H,L=.C...X..1,0;

```

It's a good idea to specify pin names in a format that is consistent with the actual pin state. In the above pin definitions, signals A and Y are active-high, while B and Z are active-low. We have chosen to indicate active-low data signals by prefixing the labels with exclamation points in the definition statement. The use of an active high variation of these signals in subsequent design statements will automatically be resolved by the software compiler.

Entry of the logic functions is next. This entry is in the form of Boolean equations, truth-table, state machine and schematic-entry formats.

Here, the Boolean equation-entry format is used to create an AND function on Y (pin 18) and an XOR function on Z (pin 19). Since Z has been defined as an active low signal, however, we will actually end up with XNOR on pin 19:

```

"**** logic equations ****
equations

```

```
Y = A & B;  
Z = A & B # !A & !B;
```

The operators used in the ABEL language are '!' for invert, '&' for the AND function and '#' for the OR function. The equations are written exactly as needed. All of the inversions for active-low inputs and outputs will be automatically resolved, a routine procedure for compiler software. Although these are simple equations, had they been complex ones that needed automatic reduction to a specific number of product terms for a given PLD, the software would have performed the reduction, as well.

When a design source file is complete, the Compile Menu options compile the source file where various input file

```

Data I/O ABEL-5 Design Environment
File Edit View Compile Optimize SmartPart PartMap Xfer Defaults Help
+-----+-----+-----+-----+-----+-----+-----+-----+
/ New /
/ Open... /
/ Insert... /
+-----+-----+-----+-----+-----+-----+-----+
/ Save /
/ Save As... /
/ Save Options /
+-----+-----+-----+-----+-----+-----+-----+
/ Print... /
/ DOS Shell /
+-----+-----+-----+-----+-----+-----+-----+
/ Save and Exit /
/ Exit /
+-----+-----+-----+-----+-----+-----+-----+

```

Figure 2. Data I/O ABEL-5 Design Environment

Using GAL Development Tools

formats are converted to equation format. Next, the Optimize Menu option minimizes the equations.

The last step of the process is to generate the JEDEC fusemap under the PartMap Menu. JEDEC, a standards organization with representatives from major semicon-

ductor companies on its committees, has approved a standard for the interchange of PLD data. The JEDEC file is used as the medium of transfer from the development computer environment to that of the hardware device programmer. Included in the file are control bits that determine the status of security cells or fuses, test

Listing 1. ABEL Documentation File

```

ABEL - Device Utilization Chart

Tutorial Using a GAL16V8
Lattice Semiconductor
-----
Module                               : 'gates'
-----
Input files:
  ABEL PLA file      : gates.tt3
  Vector file       : gates.tmv
  Device library     : P16V8AS.dev

Output files:
  Report file        : gates.doc
  Programmer load file : gates.jed

P16V8AS Programmed Logic:
-----
Y      = ( A & !B );
Z      = !( A & !B ) & !A & B ;

P16V8AS Chip Diagram:
-----
P16V8AS
-----

```

A	1	20	Vcc
B	2	19	!Z
	3	18	!Y
	4	17	
	5	16	
	6	15	
	7	14	
	8	13	
	9	12	
GND	10	11	

```

-----
SIGNATURE: N/A

```

Using GAL Development Tools

Listing 1. ABEL Documentation File (Continued)

P16V8AS Resource Allocations:				
Device Resources	Resource Available	Design Requirement	Part Utilization	Unused
Dedicated input pins	10	2	2	8 (80 %)
Combinatorial inputs	10	2	2	8 (80 %)
Registered inputs	-	0	-	-
Dedicated output pins	2	2	0	2 (100 %)
Bidirectional pins	6	0	2	4 (66 %)
Combinatorial outputs	8	2	2	6 (75 %)
Registered outputs	-	0	-	-
Two-input XOR	-	0	-	-
Buried nodes	-	0	-	-
Buried registers	-	0	-	-
Buried combinatorials	-	0	-	-

P16V8AS Product Terms Distribution:				
Signal Name	Pin Assigned	Terms Used	Terms Max	Terms Unused
Y	18	1	8	7
Z	19	2	8	6

==== List of Inputs/Feedbacks ====

Signal Name	Pin	Pin Type
A	1	INPUT
B	2	INPUT

P16V8AS Unused Resources:

Pin Number	Pin Type	Product Terms	Flip-flop Type
3	INPUT	-	-
4	INPUT	-	-
5	INPUT	-	-
6	INPUT	-	-
7	INPUT	-	-
8	INPUT	-	-
9	INPUT	-	-
11	INPUT	-	-
12	BIDIR	NORMAL 8	-
13	BIDIR	NORMAL 8	-
14	BIDIR	NORMAL 8	-
15	OUTPUT	NORMAL 8	-
16	OUTPUT	NORMAL 8	-
17	BIDIR	NORMAL 8	-

Using GAL Development Tools

vectors, and data-transmission checksums. (The JEDEC standard is available from Lattice Semiconductor upon request.) A portion of the JEDEC file for our example is reproduced here:

```
QP20* QF2194* QV8* F0*
X0*
NOTE Table of pin names and numbers*
NOTE PINS A:1 B:2 Y:18 Z:19*
L0000 10011111111111111111111111111111*
L0032 01101111111111111111111111111111*
L0256 10011111111111111111111111111111*
L2048 01000000*
L2128
111111111111111111111111111111111111*
L2192 1*
C13DA*
```

Test vectors, which indicate the stimulus and response for a PLD, serve primarily to validate the functionality of a design source file. The ABEL compiler thus simulates the source file on paper, so that, hopefully, only properly functioning patterns are ever programmed into a PLD for system debug. In our basic gates example, the source file for the simulator routine provides the expected data:

```
**** test vector definition ****
test_vectors
([A,B] -> [Y,Z]);
[0,0] -> [L,X]; "**** test AND gate ****
[0,1] -> [L,X];
[1,0] -> [L,X];
[1,1] -> [H,X];
[0,0] -> [X,H]; "**** test XNOR gate ****
[0,1] -> [X,L];
[1,0] -> [X,L];
[1,1] -> [X,H];
end
```

While some PLD manufacturers claim that test vectors are also necessary for verifying functionality of the integrated circuit after programming, Lattice E²CMOS GAL devices are fully tested and guaranteed to yield 100% all of the time. In Section 11 of this handbook, the issue of testability and how Lattice achieves this unmatched quality level is discussed in detail.

Once the JEDEC fusemap is generated, the design is ready to be programmed into a device. ABEL also generates a documentation file (.DOC) for the design which can be viewed under the View Menu option. The purpose

of the file is to provide a hard-copy documentation of the final (reduced) equations, the cell map or 'fuse plot,' and a chip-pinout diagram, if desired (see listing 1).

Example: Two-Story Elevator Controller

This example is designed to step the reader through the process of creating and implementing a logic design using GAL devices. Whether a novice or intermediate user of PLDs, the reader is encouraged to familiarize himself with this section, as well as with the applications in the GAL application notes section, which provides examples of how to implement basic functions such as decoders, shifters, multiplexers, counters, and so on.

Here we will be building a two-story elevator control unit. The function of the unit is to monitor the state of the call buttons, respond to calls for service, and display the status of the elevator by means of floor and direction displays. The operating control function requires a small state machine and a latch function, while the display logic uses only combinational circuits.

Our elevator travels between two floors. Arriving at a floor in response to a call for service, the elevator opens its doors, pauses, then closes them automatically. If the Up or Down button is pushed, the elevator travels to the other floor. A microswitch mounted on the car informs the controller that the elevator has arrived at a new floor.

Once the elevator arrives at a floor to discharge passengers, it opens its doors, pauses to let the passengers out, then closes the doors and assumes its wait position. A call for service at the floor where the elevator is resting will result in the doors being opened.

A free-running clock controls the elevator's operation, toggling every 5 to 10 seconds to allow a brief pause during each arrival and departure activity. While this slow clock rate is appropriate for the timing of the elevator doors and car movement, it is far too slow to capture a time-independent call for service. As such, a latch function that captures data instantly (actually within 25 ns) is designed using two of the GAL device macrocells.

As shown in Figure 3, the total elevator control unit uses two GAL16V8s — one to perform the actual control function, the other to handle the display.

The control of the elevator consists of two basic functions: the call-button latches and the state machine. The latches, constructed from the GAL device's available AND and OR gates (instead of using the on-chip D-type register), are instantaneous and not dependent on a clock for holding data. The truth table of the S-R latch

Using GAL Development Tools

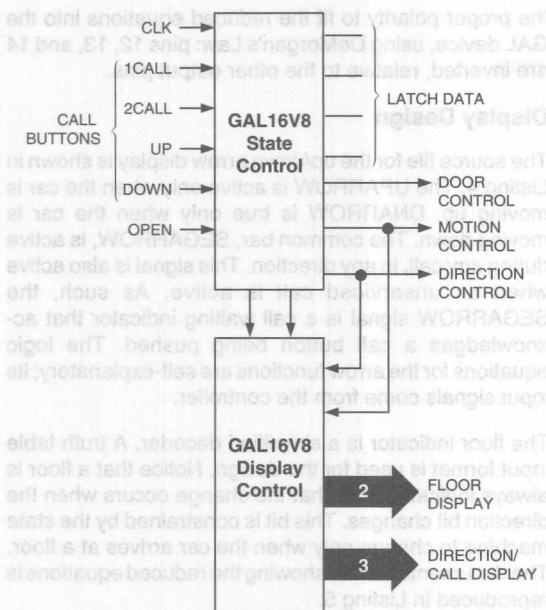


Figure 3. Block Diagram

used is shown in Figure 4. As shown in the truth table, the latch is set by applying a logic 1 to SET, and reset by applying a logic 1 to RESET. Applying a logic 0 to both inputs causes a hold state, while applying a logic 1 to both is undefined for this type of latch. The various call signals (UCALL, DCALE, OCALL, 1CALL, 2CALL) are applied to the two latches to command the elevator to travel to the requested floor.

The first step in this GAL implementation is to translate the functional operation of the elevator (described in the text in the preceding paragraphs) to a logical format. This is realized through the use of a state-transition diagram, which literally describes all the allowed stable states (on floor two, doors open, etc.) that our elevator can be in. An unacceptable state, for example, would be resting between floors.

Set	Reset	Output	Output
0	0	Hold	Hold
0	1	0	1
1	0	1	0
1	1	- Not Valid -	

Figure 4. SR Latch Truth Table

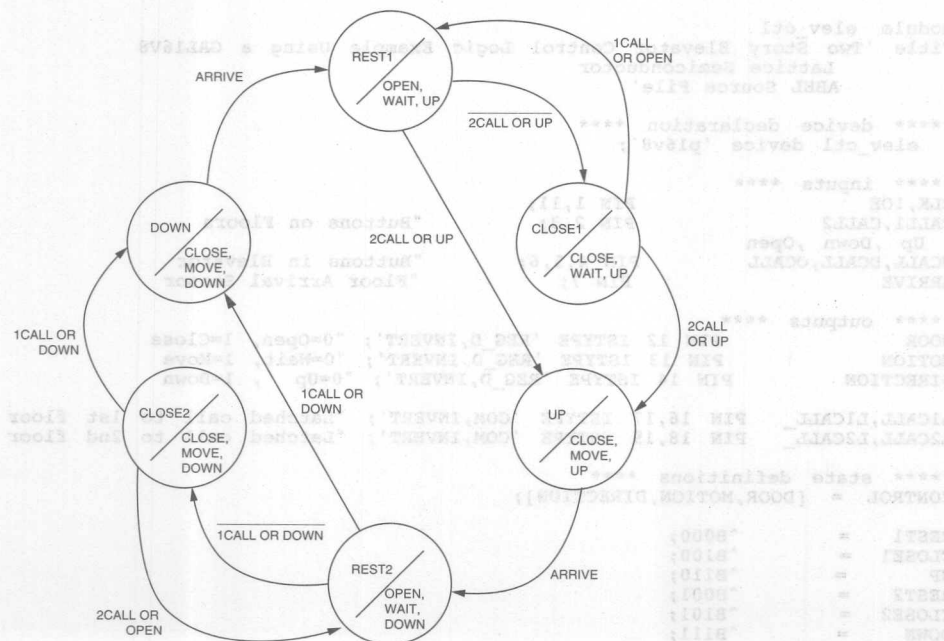


Figure 5. State-Transition Diagram

Using GAL Development Tools

Figure 5 shows the state-transition diagram. Inside each state circle, the diagram indicates the state name (top half) and the condition of each of the state variables: DOOR, MOTION, and DIRECTION. The transitions out of the state are shown with the logic level requirements to make the transition. Also shown is the destination of each of the transfers.

The latched signals L1CALL and L2CALL are used to start the states changing. The ARRIVAL input tells the car when to stop its motion. The normal wait state of the elevator is either CLOSE1 or CLOSE2 (not moving with door closed).

The information is then transferred from the transition diagram to the ABEL state-machine syntax, shown in Listing 2. Notice the use of defaults in the state syntax to indicate what state should be selected (or held) if none of the criteria for exit is met. There is also an identifiable 1-to-1 correspondence from the state transition diagram to the state-machine syntax. The portion of the documentation file which includes reduced equations is shown in Listing 3. Notice that the compiler automatically chose

the proper polarity to fit the reduced equations into the GAL device, using DeMorgan's Law: pins 12, 13, and 14 are inverted, relative to the other output pins.

Display Design

The source file for the up/down arrow display is shown in Listing 4. The UPARROW is active only when the car is moving up. DNARROW is true only when the car is moving down. The common bar, SEGARROW, is active during any call, in any direction. This signal is also active when an unserved call is active. As such, the SEGARROW signal is a call waiting indicator that acknowledges a call button being pushed. The logic equations for the arrow functions are self-explanatory; its input signals come from the controller.

The floor indicator is a simplified decoder. A truth table input format is used for the design. Notice that a floor is always indicated, and that the change occurs when the direction bit changes. This bit is constrained by the state machine to change only when the car arrives at a floor. The documentation file showing the reduced equations is reproduced in Listing 5.

Listing 2. Design Input File for Control Section

```
module elev_ctl
Title 'Two Story Elevator Control Logic Example Using a GAL16V8
      Lattice Semiconductor
      ABEL Source File'

"**** device declaration ****
elev_ctl device 'pl6v8';

"**** inputs ****
CLK,!OE          PIN 1,11;
CALL1,CALL2      PIN 2,3;
" Up ,Down ,Open
UCALL,DCALL,OCALL PIN 4,5,6;
ARRIVE           PIN 7;

"Buttons on Floors
"Buttons in Elevator
"Floor Arrival Sensor

"**** outputs ****
DOOR             PIN 12 ISTYPE 'REG_D,INVERT'; "0=Open, 1=Close
MOTION          PIN 13 ISTYPE 'REG_D,INVERT'; "0=Wait, 1=Move
DIRECTION       PIN 14 ISTYPE 'REG_D,INVERT'; "0=Up , 1=Down

L1CALL,L1CALL_   PIN 16,17 ISTYPE 'COM,INVERT'; "Latched call to 1st floor
L2CALL,L2CALL_   PIN 18,19 ISTYPE 'COM,INVERT'; "Latched call to 2nd floor

"**** state definitions ****
CONTROL = [DOOR,MOTION,DIRECTION];

REST1 = ^B000;
CLOSE1 = ^B100;
UP = ^B110;
REST2 = ^B001;
CLOSE2 = ^B101;
DOWN = ^B111;

"**** intermediate definitions ****
C,X,H,L=.C.,.X.,1,0;
```

```

FLOOR1 = DIRECTION.FB;
FLOOR2 = !DIRECTION.FB;

"**** logic equations ****
equations

L1CALL = !( L1CALL_ # CALL1 # (FLOOR1 & OCALL)
              # (FLOOR2 & DCALL));
L1CALL_ = !( L1CALL # (!DOOR.FB & !MOTION.FB & !DIRECTION.FB));
L2CALL = !( L2CALL_ # CALL2 # (FLOOR2 & OCALL)
              # (FLOOR1 & UCALL));
L2CALL_ = !( L2CALL # (!DOOR.FB & !MOTION.FB & DIRECTION.FB));

CONTROL.CLK = CLK;

"**** state machine definition ****
state_diagram CONTROL
state REST1: if (L2CALL) then UP;
              else CLOSE1;
state CLOSE1: if (L2CALL) then UP;
               else if (L1CALL) then REST1;
               else CLOSE1;
state UP: if (ARRIVE) then REST2;
           else UP;
state REST2: if (L1CALL) then DOWN;
              else CLOSE2;
state CLOSE2: if (L1CALL) then DOWN;
               else if (L2CALL) then REST2;
               else CLOSE2;
state DOWN: if (ARRIVE) then REST1;
              else DOWN;

end

```

Listing 3. Expanded Product Terms for Control Section

```

L1CALL = ( !DIRECTION.Q & !L1CALL_ & !CALL1 & !OCALL
           # DIRECTION.Q & !L1CALL_ & !CALL1 & !DCALL );

L1CALL_ = !( DIRECTION.Q & DOOR.Q & MOTION.Q
             # L1CALL );

L2CALL = ( DIRECTION.Q & !OCALL & !L2CALL_ & !CALL2
           # !DIRECTION.Q & !L2CALL_ & !CALL2 & !UCALL );

L2CALL_ = !( !DIRECTION.Q & DOOR.Q & MOTION.Q
             # L2CALL );

DOOR.D = ( !DIRECTION.Q & !DOOR.Q & MOTION.Q & !L1CALL & L2CALL
           # DIRECTION.Q & !DOOR.Q & MOTION.Q & L1CALL & !L2CALL
           # !DOOR.Q & !MOTION.Q & ARRIVE ); " ISTYPE 'INVERT'

DOOR.C = ( CLK );

MOTION.D = ( !DIRECTION.Q & MOTION.Q & !L1CALL
             # DIRECTION.Q & MOTION.Q & !L2CALL
             # !DOOR.Q & !MOTION.Q & ARRIVE ); " ISTYPE 'INVERT'

MOTION.C = ( CLK );

DIRECTION.D = ( DIRECTION.Q & MOTION.Q
                # !DIRECTION.Q & !DOOR.Q & !MOTION.Q & ARRIVE
                # DIRECTION.Q & !DOOR.Q & !ARRIVE ); " ISTYPE 'INVERT'

DIRECTION.C = ( CLK );

```

Using GAL Development Tools

Listing 4. Design Input File for Display Section

```

module elev_dsp
Title 'Two Story Elevator Display Logic Example Using a GAL16V8
      Lattice Semiconductor
      ABEL Source File'

"**** device declaration ****
elev_dsp device 'pl6v8';

"**** inputs ****
L1CALL,L2CALL          PIN 2,3;          "Call Status
MOTION,DIRECTION       PIN 4,5;          "Control Status

"**** outputs ****
UPARROW                PIN 12 ISTATE 'COM,INVERT'; "
SEGARROW               PIN 13 ISTATE 'COM,INVERT'; "
DNARROW               PIN 14 ISTATE 'COM,INVERT'; "

SEG1,SEG2,SEG12 PIN 15,16,17 ISTATE 'COM,INVERT'; "LED Segment Drivers

"
"      segment display diagram
"
"      2
"      |
"      | 12
"      |
"      2
"      |
"      | 1
"      |
"      2
"

"**** intermediate definitions ****
C,X,H,L=.C.,.X.,1,0;

"**** logic equations ****
equations

UPARROW = MOTION & DIRECTION;
SEGARROW= L1CALL # L2CALL;
DNARROW = MOTION & !DIRECTION;

"**** truth table definition ****
truth_table
([DIRECTION, MOTION] -> [SEG1, SEG2, SEG12]);
[ 0 , 0 ] -> [ 1 , 0 , 1 ];
[ 0 , 1 ] -> [ 1 , 0 , 1 ];
[ 1 , 0 ] -> [ 0 , 1 , 1 ];
[ 1 , 1 ] -> [ 0 , 1 , 1 ];

end

```

Listing 5. Expanded Product Terms for Display Section

```

UPARROW = ( MOTION & DIRECTION );
SEGARROW = (!(L1CALL & !L2CALL));
DNARROW = ( MOTION & !DIRECTION );
SEG1     = !( DIRECTION );
SEG2     = !( !DIRECTION );
SEG12    = !(0);

```

GAL Development Support

Lattice Semiconductor recommends the use of qualified programming equipment when programming Lattice devices. Lattice works with several programming manufacturers to insure that there is cost effective equipment available. We have approved programmers in each of the following categories:

- Low Cost GAL Only Programmers
- Mid Range 28-pin Programmers
- Full Universal Programmers
- Production Programming Equipment

Lattice conducts a very stringent qualification procedure, which includes a complete evaluation of the programming, verification and load algorithms; verification of critical pulse widths and voltage levels, along with a complete yield analysis. The result is the best programming yields in the industry and a guarantee of 100% programming yields to customers using qualified programming equipment. Below are the third-party programmers which are qualified to program Lattice devices.

For a current listing of Lattice qualified programmers, please call Lattice's Literature Distribution Department (Tel: 503-693-0287; FAX: 503-681-3037).

Qualified Programmers

Vendor	Programmer
Data I/O	Autosite
	Unisite
	3900
	2900
	29B
	60A/H
Logical Devices	Allpro 88
	Allpro 40
Stag	System 3000
	ZL30B & ZL30A
	Quasar-U84 & Quasar-U40
System General	TURPRO-1 & TURPRO-1/FX
	SGUP-85A
SMS Microcomputer	Sprint Expert
	Sprint Plus
BP-Microsystems	BP-1200
	PLD-1128 & CP-1128
	PLD-1100
Advin	Pilot-U84 & Pilot-U40
	Pilot-U256 & Pilot-168
	Pilot-GL & Pilot-GCE

Logic Compiler Support

Vendor	Logic Compiler
Accel Tech.	Tango PLD
Cadence	PIC Designer Composer
	PIC Designer Concept
Data I/O	ABEL
ISDATA	LOG/IC
Logical Devices	CUPL
Mentor Graphics	PLSynthesis II
Minc	PLDesigner-XL
OrCAD	OrCAD PLD
Omaton	Schema-PLD
Viewlogic	ViewPLD

GAL Development Support

PROGRAMMER/COMPILER VENDORS

Accel Technologies

6825 Flanders Dr.
San Diego, CA 92121
Tel: (619) 554-1000
FAX: (619) 554-1019

Advin Systems

1050-L Duane Ave
Sunnyvale, CA 94086
Tel: (408) 243-7000
FAX: (408) 736-2503

BP Microsystems

1000 N Post Oak Road
Houston, TX 77055-7237
Tel: (713) 688-4600
1-800-225-2102

FAX: (713) 688-0902

BBS: (713) 688-9283

Cadence Design Systems

555 River Oaks Parkway
San Jose, CA 95134
Tel: (408) 943-1234
FAX: (408) 943-0513

Data I/O Corp.

10525 Willows Road N.E.
P.O. Box 97046
Redmond, WA 98073-9746
Tel: (206) 881-6444
Tel: 1-800-247-5700
FAX: (206) 882-1043

In Europe contact:

Data I/O Corp.

Tel: +31 (0) 20-6622866

In Japan contact:

Data I/O Corp.

Tel: (03) 432-6991

ISDATA GmbH

Haid-und-Neu-Straße 7
7500 Karlsruhe 1
Germany
Tel: 0721-693092
FAX: 0721-174263

In the U.S. contact

ISDATA Inc.

Tel: (408) 373-7359

FAX: (408) 373-3622

Logical Devices

692 South Military Trail
Deerfield Beach, FL 33442
Tel: (305) 428-6868
FAX: (305) 428-1181

Mentor Graphics

8005 S.W. Boeckman Rd.
Wilsonville, OR 97070
Tel: (503) 685-7000
FAX: (503) 685-1204

Minc Incorporated

6755 Earl Dr.
Colorado Springs, CO 80918
Tel: (719) 590-1155
FAX: (719) 590-7330

Omation

801 Presidential
Richardson, TX 75081
Tel: (214) 231-5167
FAX: (214) 783-9072

OrCAD Systems Corp.

3175 N.W. Alcielek Dr.
Hillsboro, OR 97124
Tel: (503) 690-9881
FAX: (503) 690-9891

SMS Micro Systems

IM Grund 15
D-7988 Wangen
Germany
Tel: (49) 7522-5018
FAX: (49) 7522-8929
In the U.S. contact:
SMS North America, Inc.
16522 N.E. 135th Pl.
Redmond, WA 98052
Tel: (206) 883-8447
FAX: (206) 883-8601

Stag Microsystems

Martinfield
Welwyn Garden City
Herts AL7 1JT
United Kingdom
Tel: 011-44-707-332148
FAX: 011-44-707-371503
In the U.S. contact:
Stag Microsystems
1600 Wyatt Dr.
Santa Clara, CA 95054
Tel: (408) 988-1118
FAX: (408) 988-1232

System General

3Fl., No. 1, Alley 8, Lane 45
Bao Shing Rd.
Shin Dian
Taipei, Taiwan R.O.C.
Tel: 886-2-9173005
FAX: 886-2-9111283
In the U.S. contact:
System General
510 S. Park Victoria Dr.
Milpitas, CA 95035
Tel: (408) 263-6667
FAX: (408) 262-9220

Viewlogic Systems

293 Boston Post Rd. West
Marlboro, MA 01752
Tel: (508) 480-0881
FAX: (508) 480-0882

Copying PAL, EPLD & PEEL Patterns Into GAL Devices

Introduction

The generic/universal architectures of Lattice GAL devices are able to emulate a wide variety of PAL, EPLD and PEEL devices. GAL devices are direct functional and parametric replacements for most PLD device architectures. To use GAL devices in place of other PLD types, some conversion of the original device pattern may be needed. This conversion is not difficult, and can be accomplished at either the design or manufacturing level. The following sections describe several techniques available to convert PAL, EPLD and PEEL device patterns to Lattice GAL device patterns.

Cross Programming: GAL16V8 and GAL20V8

The GAL16V8 and GAL20V8 devices replace most standard 20-pin and 24-pin PAL devices. To simplify the conversion process, Lattice has worked with programmer hardware manufacturers to provide the ability to program GAL devices directly from existing PAL JEDEC files, or master PAL devices. Lattice qualified programmers can automatically configure the architecture of a GAL device to emulate the source PAL device.

To provide a conceptual framework for the conversion from PAL devices to GAL devices, a mythical device known as a RAL device was created. A RAL device is simply a GAL device configured to emulate a PAL. There is a one-to-one correspondence between the name of a PAL device and that of a RAL device. For example, a RAL16L8 is simply a GAL16V8 configured as a PAL16L8. Some programmers list the RAL device types as choices for cross-programming, while others specifically state that a cross-programming operation is to be performed using a PAL device type as the architecture type. Other programmers list devices such as a Lattice 16L8. Even though Lattice does not make a 16L8 device, choosing this selection allows the programmer to accept a 16L8 JEDEC file, and will program a GAL16V8 device to emulate a PAL16L8.

To program a GAL16V8 or GAL20V8 device from an existing PAL JEDEC file, simply select the appropriate device code (either RAL type, or PAL type to cross-program from), then download the PAL JEDEC file to the programmer. Insert the appropriate GAL device that can directly emulate the PAL device (according to the chart in the GAL16V8 or GAL20V8 datasheets). The programmer will automatically configure the GAL device to emulate the PAL device during programming. The resulting GAL device is 100% compatible with the original PAL device.

A GAL device may also be programmed from a master PAL device by reading the pattern of the master PAL into the programmer memory, then selecting the appropriate RAL device or PAL type to cross-program from. The GAL device can then be programmed from the programmer memory.

Cross Programming: GAL22V10/GAL20RA10

The GAL22V10 and GAL20RA10 are direct replacements for bipolar PAL devices, and are JEDEC fuse map compatible with these industry standard devices. To program a GAL22V10 or GAL20RA10 device from an existing PAL JEDEC file, simply select the appropriate GAL device code, then download the PAL JEDEC file to the programmer. The resulting GAL device is 100% compatible with the original PAL.

GAL devices also may be programmed from Master PAL devices by reading the pattern of the Master PAL into the programmer memory, then selecting the appropriate GAL device code. The GAL device can then be programmed from the programmer memory.

The GAL22V10 and GAL20RA10 also can store a User Electronic Signature (see the datasheets on these devices for more information). To use this feature, the JEDEC file must contain this information. To add the signature data to the JEDEC map, use the PALtoGAL conversion utility or recompile the source equations for a Lattice GAL device instead of a generic 22V10 type. Many programmers list two device types to differentiate between the two types of JEDEC files, and list both a GAL22V10 and a name such as GAL22V10-UES or GAL22V10-ES. Other programmers allow both types of JEDEC files to be accepted, and simply don't program the Signature fuses if they are not present in the file.

Cross Programming: GAL20XV10

The GAL20XV10 can be configured as a direct replacement for bipolar PAL20X10, 20X8, 20X4, and 20L10 devices. Many programmers provide cross-programming support similar to that provided for the GAL16V8/GAL20V8 devices. This allows the use of existing PAL device files to program the GAL20XV10 to emulate the PAL devices. The PALtoGAL conversion software (described below) also supports conversion of the PAL JEDEC files to a functionally equivalent GAL device file.

Copying PAL, EPLD & PEEL Patterns into GAL Devices

PALtoGAL Conversion Utility Software

Lattice has created a software utility that will convert an existing PAL device JEDEC file to the appropriate GAL device JEDEC format. Called PALtoGAL, this software utility can be used to convert PAL device files to GAL device files, add/or change the User Electronic Signature without changing device functionality, and reformat existing GAL JEDEC files for readability.

Since a few programmable logic devices have features that a GAL device cannot exactly emulate, the PALtoGAL utility will clearly describe the incompatibility but will not create an output file. GAL devices programmed using files converted by PALtoGAL will be 100% compatible with the original logic device. PALtoGAL is just another method of cross-programming, and should produce the same results as using a programmer. The advantage is that a full GAL device JEDEC map is created, meaning that the appropriate GAL device may then be selected on the programmer, which may simplify the manufacturing flow. Also, the PALtoGAL conversion software provides conversions that programmers do not.

A copy of the PALtoGAL conversion utility software can be obtained through your local Lattice representative, or

by contacting the GAL Applications Hotline at 1-800-FASTGAL (327-8425) or (503) 693-0201. The software also may be downloaded from Lattice's Electronic Bulletin Board at (503) 693-0215; the file name is "PALTOGAL.EXE".

Software Compiler Conversion

If the equation source file is available for the PAL device, it can be converted by re-compiling using a suitable logic compiler that supports GAL devices. If there are any device incompatibilities (there shouldn't be in most cases), the compiler will describe the errors. The output of the compiler will be a GAL JEDEC file that can be used to program a GAL device directly. The resulting GAL device will be 100% functionally compatible with the original device.

Suitable logic compilers are listed in the Development Tools section. If additional questions arise, contact your compiler manufacturer or a Lattice Applications Engineer by calling the GAL Applications Hotline at 1-800-FASTGAL or (503) 693-0201.

The GAL20K10 and GAL20K10 also can store a User Electronic Signature (see the data sheets on these devices for more information). To use this feature, the JEDEC file must contain this information. To add the signature data to the JEDEC map, use the PALtoGAL conversion utility to compile the source equations for a Lattice GAL device instead of a generic 22V10 type. Many programmers list two device types to differentiate between the two types of JEDEC files, and list both a GAL20K10 and a name such as GAL20K10-ES. Other programmers show both types of JEDEC files to be accepted, and simply don't program the Signature fuses if they are not present in the file.

Cross Programming: GAL20K10

The GAL20K10 can be configured as a direct replacement for bipolar PAL20K10, 20K4, and 20L10 devices. Many programmers provide cross-programming support which is provided for the GAL16V8 and GAL20K10. This allows the use of existing PAL device files to program the GAL20K10 to emulate the PAL device. The PALtoGAL conversion software (described below) also supports conversion of the PAL JEDEC files to a functionally equivalent GAL device file.

To provide a conceptual framework for the conversion from PAL devices to GAL devices, a mythical device known as a RAL device was created. A RAL device is simply a GAL device configured to emulate a PAL. There is a one-to-one correspondence between the name of a PAL device and that of a RAL device. For example, a RAL16L8 is simply a GAL16V8 configured as PAL16L8. Some programmers list the RAL device types as choices for cross-programming, while others specifically state that a cross-programming operation is to be performed using a PAL device type as the architecture type. Other programmers list devices such as Lattice 16L8. Even though Lattice does not make a 16L8 device, choosing this selection allows the programmer to select a 16L8 JEDEC file, and will program a GAL16V8 device to emulate a PAL16L8.

To program a GAL16V8 or GAL20V8 device from an existing PAL JEDEC file, simply select the appropriate device code (either RAL type or PAL type) to cross-program from, then download the PAL JEDEC file to the programmer. Insert the appropriate GAL device that can directly emulate the PAL device (according to the chart in the GAL16V8 or GAL20V8 data sheet). The programmer will automatically configure the GAL device to emulate the PAL device during programming. The resulting GAL device is 100% compatible with the original PAL device.

Section 1: Introduction

Section 2: ispLSI and pLSI Architecture Overview

Section 3: ispLSI and pLSI Development Tools

Section 4: ispLSI and pLSI Application Notes

Section 5: GAL Architecture Overview

Section 6: GAL Development Tools

Section 7: GAL Application Notes

Zero-Power GAL Devices	7-1
The GAL16VP8 and GAL20VP8	7-5
The GAL18V10 Advantage	7-7
GAL20RA10: Programmable Clocks Improve System Performance	7-11
GAL6002 Designs Using ABEL and CUPL	7-13
GAL6002: 4-to-1 RS232 Port Multiplexer	7-17
VME Bus Arbitration Using a GAL22V10	7-21
GAL16VP8/20VP8: Bus Arbitration Circuit	7-25
GAL20XV10: Data Block Transfer Address Detector	7-29
GAL26CV12: Programmable Frequency Divider	7-33

Section 8: In-System Programmable Generic Digital Switch (ispGDS)

Section 9: Design Techniques

Section 10: Article Reprints

Section 11: Technology, Quality, and Reliability Overview

Section 12: General Section

Section 1: Introduction	
Section 2: lqL81 and pL81 Architecture Overview	
Section 3: lqL81 and pL81 Development Tools	
Section 4: lqL81 and pL81 Application Notes	
Section 5: GAL Architecture Overview	
Section 6: GAL Development Tools	
Section 7: GAL Application Notes	
Zero-Power GAL Devices	7-1
The GAL16V8 and GAL20V8	7-8
The GAL18V10 Advantage	7-7
GAL20RA10: Programmable Clocks Improve System Performance	7-11
GAL8002 Design Using ABEL and CUP	7-13
GAL8002: 4-to-1 R8232 Port Multiplexer	7-17
VME Bus Arbitration Using a GAL22V10	7-21
GAL16V8/20V8: Bus Arbitration Circuit	7-25
GAL20XV10: Data Block Transfer Address Detector	7-29
GAL28CV12: Programmable Frequency Divider	7-33
Section 8: In-System Programmable Generic Digital Switch (ispGDS)	
Section 9: Design Techniques	
Section 10: Article Reprints	
Section 11: Technology, Quality, and Reliability Overview	
Section 12: General Section	



Zero-Power GAL Devices

Introduction

The Zero Power GAL16V8 and GAL20V8 families of devices provide the highest speed and lowest power combination available in the PLD market. They operate at a Tpd of 12ns and an Fmax of 83.3MHz, while at a maximum Icc of 55mA and an Isb (standby current) of 50µA typical. These zero power PLDs have industry standard GAL16V8 and GAL20V8 architectures, and are manufactured with Electrically Erasable CMOS (E²CMOS) technology, offering 100% programmability, functionality and testability.

This family offers two zero power options for each architecture: An Input Transition Detection version and a Dedicated Power-Down Pin version. The GAL16V8Z and the GAL20V8Z use input transition detection to enter into the zero power mode—if there are no input transitions for a specified interval, the device powers down. The GAL16V8ZD and GAL20V8ZD enter the zero power mode by using a dedicated power-down pin which takes the place of a logic input.

Since these zero power E²CMOS PLDs have the same architectures as the GAL16V8 and GAL20V8, they can be used in similar applications. DMA control, state machines, and other standard 16/20V8 applications that become very power conscious when implemented in battery powered systems are ideal for the zero power GAL families. Also, zero power GAL devices help reduce overall system cost: they allow the user to specify smaller, less expensive power supplies and may even allow the system to be implemented without cooling fans.

This application note describes the timing parameters and architectural features of the zero power GAL devices and describes a few applications in which these devices are particularly well suited.

GAL16V8Z and GAL20V8Z

Timing Parameters

Figure 1 illustrates the timing parameters of the GAL16/20V8Z devices associated with standby mode. The GAL16/20V8Z devices enter into standby mode if there is a lack of activity on their inputs or I/Os for a period of time greater than Tas (140 ns). This makes it possible for the GAL16/20V8Z devices to automatically go into standby mode when the system or any section of the system

containing the zero power devices goes dormant. Since the GAL16/20V8Z devices may or may not be in a dormant state at any given time, they have two different propagation delays depending on which state they are in. The first propagation delay is the time taken if the devices are not in standby mode: 12ns maximum for 12ns Tpd rated devices. The second propagation delay, as indicated in figure 1, is for the first transition after sleep mode: 25ns (Tsa+Tpd) maximum for 12 ns devices.

Current Consumption

Figure 1 can be redrawn as shown in figure 2 (titled "Power Timing Waveforms") to better illustrate the calculation of average Icc current consumed by the devices. Figure 2 shows that the Icc current over time has a periodicity or cycle time (Tsr). There are two separate cases of current consumption to consider.

Case 1: If the transition timing on the I/Os or inputs is less than Tas+Tpd, the GAL16/20V8Z devices will not go into standby mode, and current consumption will be the Icc active specification (55mA).

Case 2: If the time between any successive transitions is greater than Tas+Tpd, then the GAL16/20V8Z devices will go into standby mode. If the input signals are repetitive in nature, the average Icc current consumed will be given by the following equation:

$$I_{cc}(\text{Average}) = ((I_{cc} \text{ Active}) \times (T_{as} + T_{pd}) / T_{sr}) + (I_{cc} \text{ Standby}) \times (1 - ((T_{as} + T_{pd}) / T_{sr}))$$

Architectural Features

The GAL16/20V8Z devices incorporate transition detection and timing circuitry on the inputs and I/Os to determine if the devices are to enter standby mode. A unique input buffer makes the inputs and I/Os less susceptible to noise that might be present, thus keeping the devices from leaving standby mode unnecessarily. The circuitry used can be thought of as a current barrier that is not of sufficient magnitude to interfere with normal logic operations; inputs or I/Os must either source or sink typically 30 µA of current for a state transition to occur.

The current barrier is useful as it functions as either a pull-up or pull-down resistor depending on the state of the input. Since the drive direction of these active resistors reverses when the signal passes through the threshold switching region, a monotonic input signal is latched. The

Zero-Power GAL Devices

Figure 1. ITD Standby Power Timing Waveform

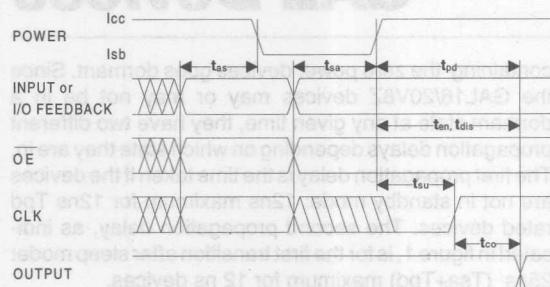
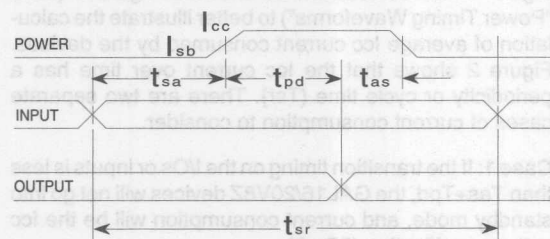


Figure 2. Power Timing Waveforms



current barrier also helps maintain the input logic level if the driver of the input goes into the high impedance state. This is valuable as it keeps the inputs out of the threshold voltage region.

Applications

Use of GAL16/20V8Z devices in systems with fast, free running clocks.

Systems that require the operation of a PLD device for small periods of time and are driven by fast, free running clocks can become energy misers through the use of GAL16/20V8Z devices. These devices can be selectively placed into standby mode by gating off of the system clock, while still retaining the capability to respond to asynchronous signals.

Figure 4 gives an example clock interface to GAL16/20V8Z devices that allows simple gating off of the system clock without false clocking of the devices. This clocking scheme supports high performance systems that consume minimal amounts of power. Clocking of the GAL16/20V8Z devices can be turned off, yet the devices will still be able to respond to asynchronous inputs. This is impossible for dedicated power-down pin devices. The GAL16/20V8Z clock needs to be disabled while the source clock is in the high state to avoid false clocking:

this has the effect of keeping the GAL16/20V8Z device clock frozen in the high state. However, the clock gating signal may be enabled at any time during the clock cycle. Note that the user must not violate setup and hold times to ensure proper operation of the circuits.

Use of GAL16/20V8Z devices in systems with slow, free running clocks.

For slow clock applications, where clock edge transitions are more than 140ns apart, circuits such as those shown in figure 5 may be used as clock drivers to GAL16/20V8Z devices. Either circuit produces a clock signal that is rising edge sensitive; when a rising edge occurs, a negative going pulse is generated. Timing waveforms showing this are in figure 6. Again, the user of such circuits is encouraged to look at the setup and hold time requirements to ensure proper circuit operation.

This negative going pulse has a falling edge which causes the GAL16/20V8Z device to go from standby mode to active mode, so that the rising edge of the negative pulse clocks the device. This circuit allows the system to be clocked down to 0 MHz. The width of the negative clock pulse, which is controlled by the delay introduced by the string of inverters, should have a minimum pulse width of $T_{sa} + T_{su}$. It should be noted that the active rising clock edge is now delayed by the circuitry in its path. This should be taken into account in the system's timing analysis.

Figure 3. DPP Timing Waveform

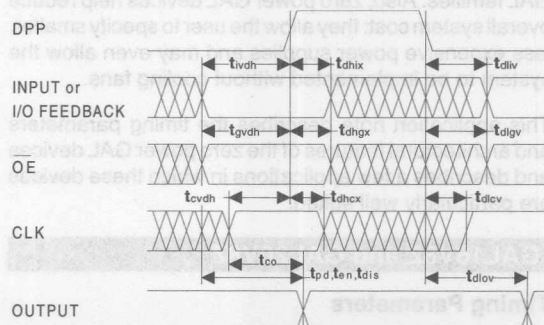
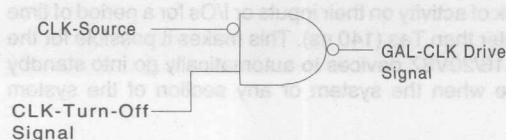


Figure 4. High Frequency Clock Drive Circuit



GAL16V8ZD and GAL20V8ZD

Timing Parameters

The GAL16/20V8ZD zero power devices have a dedicated power-down pin (DPP). When this pin is placed into the active high (DPP=H) state, the GAL16/20V8ZD devices go into standby mode, and current consumption (I_{cc}) decreases from 55mA to 100 μ A. The DPP pin is not available for logic input into the AND array, as it is with the GAL16/20V8Z devices. Since the GAL16/20V8ZD devices suspend their state when in standby mode, there are also setup and hold times on inputs to the devices with respect to the rising edge of the DPP signal. Figure 3 shows the AC timing diagrams for standby mode on the GAL16/20V8ZD devices.

The setup timing parameter for combinatorial signals is T_{vdh} , and the hold time is T_{dhx} . If these timing parameters are met, then a standard propagation delay will apply to the output. If the DPP pin is held high, the state of the output will be preserved independent of any changes to the inputs or I/Os, including the clock signal. The setup and hold times for the clock signal are T_{cvdh} and T_{dhcx} , respectively. Note that in all cases, if the timing parameters are violated then proper operation of the devices should not be expected.

Architectural Features

Operation of the DPP pin may be thought of as performing a "Chip Enable" type of function, with the exception that all of the I/Os retain the same configuration and drive state they had before going into standby mode.

Similarly, the DPP pin may be compared to a "Latch Enable" type of function, where taking the DPP pin to the high state will in effect freeze the state of the device, including all input and I/O pins. Therefore, latches on the PCB may be replaced by the functionality of the power-down pin. This double functionality of the GAL16/20V8ZD devices is beneficial as the overall part count of the system may be reduced, resulting in the benefits of lower cost, reduced board space and greater reliability.

Applications

The DPP lets the microprocessor or controller explicitly control how the battery or other energy sources are used. This is especially valuable when used in a system with other power-down devices and a global power-down control signal; power consumption can be reduced to nearly nothing. Also, the dedicated power-down pin can be effectively used for power management at critical times, such as during power drop out, when the processor or controller may be getting its power from the residual energy left in the electrolytic capacitors of the power supply. Since exercising the power-down pin puts

Figure 6. Low Frequency Clock Drive Circuit Timing Waveforms

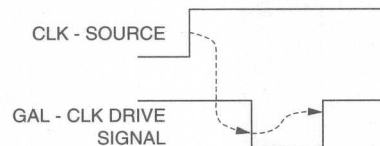
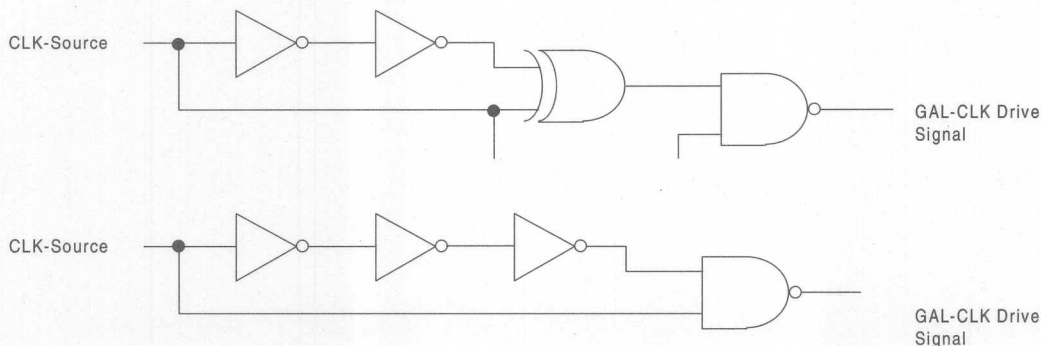


Figure 5. Low Frequency Clock Drive Circuits



Zero-Power GAL Devices

the GAL16/20V8ZD devices in standby mode and a frozen state, by placing the power-down pin into standby mode (DPP=L), resuming operation from where the GAL16/20V8ZD operation was suspended can be easily accomplished. Therefore, the processor/controller can effectively execute a power-down routine when power drops out, suspend the routine when the power critical time has ended, and allow the GAL16/20V8ZD devices to be in the same state as before power drop out.

Conclusion

Having introduced the GAL16/20V8Z and GAL16/20V8ZD devices, Lattice makes available extremely low power, high performance components. Since power savings at the component level translates to power savings at the system level, existing high performance systems can now be converted to power misers without even changing JEDEC files.

Figure 6. Low Frequency Clock Drive Circuit Timing Waveforms



GAL16V8ZD and GAL20V8ZD

Timing Parameters

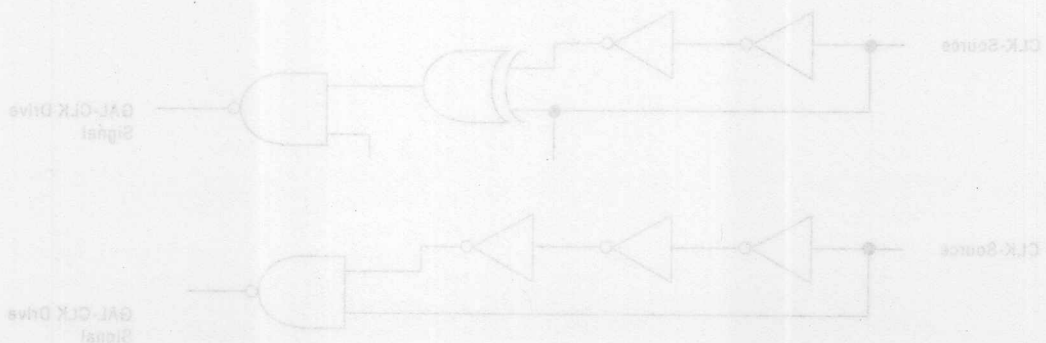
The GAL16/20V8ZD zero power devices have a dedicated power-down pin (DPP). When this pin is placed into the active high (DPP=H) state, the GAL16/20V8ZD devices go into standby mode, and current consumption (Icc) decreases from 65mA to 100µA. The DPP pin is not available for logic input into the AND array, as it is with the GAL16V8Z devices. Since the GAL16/20V8ZD devices suspend their state when in standby mode, they are also setup and hold times on inputs to the devices with respect to the rising edge of the DPP signal. Figure 8 shows the AC timing diagrams for standby mode on the GAL16/20V8ZD devices.

The setup timing parameter for combinational signals is T_{setup}, and the hold time is T_{hold}. If these timing parameters are met, then a standard propagation delay will apply to the output. If the DPP pin is held high, the state of the output will be preserved independent of any changes to the inputs or I/Os, including the clock signal. The setup and hold times for the clock signal are T_{setup} and T_{hold}, respectively. Note that in all cases, if the timing parameters are violated then proper operation of the device should not be expected.

Architectural Features

Operation of the DPP pin may be thought of as performing a "Chip Enable" type of function, with the exception that all of the I/Os retain the same configuration and drive state they had before going into standby mode.

Figure 5. Low Frequency Clock Drive Circuits





The GAL16VP8 and GAL20VP8

Introduction

Lattice's two high-drive "VP" series devices, the GAL16VP8 and GAL20VP8, are based upon the industry standard GAL16V8 and GAL20V8 architectures, adding programmable output configuration for higher drive capability. The "VP" series has programmable output buffers that can be configured to either open-drain or totem-pole outputs. Their output buffers can be independently programmed by setting the appropriate control bits in the architectural array. Additionally, their input buffers contain Schmitt trigger inputs for greater noise immunity and active pull-up resistors on all inputs and I/Os.

The GAL "VP" high drive adds value to the popular GAL16V8 and GAL20V8 product line. The advantages of the "VP" series include:

- **Higher Output Drive Current**
 - Low-Level Output Current
 - I_{ol} = 64 mA vs. 24mA
 - High Level Output Current
 - I_{oh} = -32 mA vs. -3.2mA
- **Schmitt Trigger Input Buffers** – Schmitt trigger input buffers with 200mV of hysteresis between positive and negative input transitions. The Schmitt trigger inputs offer improved noise immunity during switching transitions, especially on the clock input. Hysteresis prevents double clocking when non-monotonic rise and fall times are present.
- **Programmable Output Buffers** – Two independent types of output buffers can be programmed for each OLMC (Output Logic Macro Cell). A combination of open-drain and totem-pole outputs can be used. For example, four totem-pole outputs for interfacing I/O functions and four open-drain outputs for bus interfacing with pull-up resistors. Any mixed combination of output buffers can be used since each output macrocell contains a dedicated fuse to assign the configuration of the output buffer.
 - Totem-pole output for standard high-drive interfacing to external logic systems with heavy capacitive loading. This configuration has V_{ol} and V_{oh} levels that are standard TTL-level data sheet values, V_{ol}=0.5 V max., V_{oh}=2.4 V min.

- Open-drain output for bus interfacing and arbitration circuits. Low logic level V_{ol} has the standard TTL-level data sheet value, V_{ol} = .5 V max. High logic level V_{oh} is set from external pull-up resistors and is a function of the external loading.

These advantages allow the designer greater flexibility when interfacing to bus and memory logic.

The series offers 64mA I_{ol} output drive for driving heavy capacitive loads associated with memory elements, such as those on data buses and back plane type systems.

One of the advantages of using high drive programmable logic for interfacing is that it eliminates the need for 74XX240 type drivers that are used in conjunction with decoding logic as a multiple device solution. In many microprocessor applications, decoding logic is used to decode address space for I/O or memory, then these signals are fed to bus driver components, such as the 74XX240 series, to drive heavy loads on busses or back planes.

Using the Lattice "VP" high drive series allows the designer to accomplish this with a single chip solution. The GAL20VP8 or GAL16VP8 devices are used to decode the address space needed and the appropriate output configuration is chosen to supply the drive capability needed to interface with the system.

Programming the 16/20VP8

Development systems such as ABEL from Data I/O or CUPL from Logical Devices support both the GAL16VP8 and the GAL20VP8 with compiler support. Logic equations and syntax remain the same as with standard GAL16V8 and GAL20V8 devices. There are three possible modes that are used for different OLMC configurations along with the output drive configuration mode: Registered, Complex and Simple. Each of these modes is set according to the logic functions implemented in the source or design file. The only additional information needed in the source file is the configuration of the output buffers. The output buffer configuration is set with a dedicated architecture fuse for each OLMC, architecture fuse AC2.

- AC2 = 1 Defines totem-pole output.
- AC2 = 0 Defines open-drain output.

The GAL16VP8 and GAL20VP8

Each OLMC has an associated set of architecture fuses; the fuses SYN, AC1, AC0 will be set by the compiler software for the appropriate OLMC mode (Registered, Complex or Simple).

Each output also contains an AC2 fuse. The following is a list of AC2 fuse locations:

GAL16VP8	GAL20VP8
Output 19 (AC2 = 2194)	Output 22 (AC2 = 2706)
Output 18 (AC2 = 2195)	Output 21 (AC2 = 2707)
Output 17 (AC2 = 2196)	Output 20 (AC2 = 2708)
Output 16 (AC2 = 2197)	Output 19 (AC2 = 2709)
Output 14 (AC2 = 2198)	Output 17 (AC2 = 2710)
Output 13 (AC2 = 2199)	Output 16 (AC2 = 2711)
Output 12 (AC2 = 2200)	Output 15 (AC2 = 2712)
Output 11 (AC2 = 2201)	Output 14 (AC2 = 2713)

These fuses must be set using the compiler software. The following statements for ABEL and CUPL compilers show how to implement the programmable output buffers.

Example 1. ABEL Example for the GAL16VP8-15LP (Setting the Output Driver Fuses)

```
Module TEST1
TITLE 'This is an example for the GAL16VP8-15LP that sets the output driver fuses';

TEST1 Device 'P16VP8';
" Note: The GAL16VP8-15LP is a center-pin device. Ground = Pin 15 , Vcc = Pin 5
" Pin Assignments

IN1, IN2, IN3, IN4, IN6, IN7, IN8, IN9      Pin 1,2,3,4,6,7,8,9;
OUT11, OUT12, OUT13, OUT14                 Pin 11,12,13,14;
OUT16, OUT17, OUT18, OUT19                 Pin 16,17,18,19;

" Use the FUSES statement to individually set AC2 fuses for output configurations.

FUSES [2194..2197] = [1,1,1,1];      "Set output pins 19,18,17,16 to totem-pole
FUSES [2198..2201] = [0,0,0,0];      "Set output pins 14,13,12,11 to open-drain

EQUATIONS

OUT19 = IN1 & !IN2 & !IN4;      " Pin 19 configured as totem-pole.
OUT18 = !IN1 & !IN2 & !IN3;      " Pin 18 configured as totem-pole.
OUT17 = IN4 & IN6 & !IN8;        " Pin 17 configured as totem-pole.
OUT16 = IN6 & IN7 & !IN8;        " Pin 16 configured as totem-pole.

OUT14 = !IN3 & !IN8;            " Pin 14 configured as open-drain.
OUT13 = IN2 & IN7;              " Pin 13 configured as open-drain.
OUT12 = !IN2 & !IN4 & IN7;        " Pin 12 configured as open-drain.
OUT11 = !IN4 & IN6 & IN8;        " Pin 11 configured as open-drain.
end
```




The GAL18V10 Advantage

Introduction

Although the GAL16V8 is able to replace a number of different standard PLDs, such as the common PAL16L8 and PAL16R8, there are times when a designer needs more flexibility than standard 20-pin PLDs offer. Moving the PLD design to the next package size means using a 24-pin DIP or 28-pin PLCC package. Often the increase in functionality does not justify the increase in package size.

This application brief describes the most common limitations of a standard 20-pin PLD and how the GAL18V10's unique architecture allows the designer much greater functionality while maintaining the same 20-pin package. In addition, the architecture of the 18V10 is virtually the same as the industry-standard 22V10 device, which means that learning a new device architecture is not necessary.

More Inputs

As with the GAL22V10, the Lattice GAL18V10 macrocell structure allows for greater flexibility than the common 20-pin PAL-type devices. Whereas the GAL16V8, because of its exact emulation of many PAL devices, must limit the I/O pins that can have feedback or be configured as inputs, the 18V10 has no such limitations. Every one of the I/O pins on the GAL18V10 can be configured as registered or combinatorial, has feedback capability, and

can be configured as a dedicated input or dynamic I/O pin.

GAL16V8 Emulation Mode	No Feedback or Input
Complex Mode (16L8)	pins 12, 19
Simple Mode (16H4, etc.)	pins 15, 16

More Outputs

As the name suggests, the GAL18V10 has a total of 10 possible outputs. In cases where more than eight outputs are needed on the standard PLD, a GAL18V10 is an ideal replacement. One demonstration of the additional capability of the GAL18V10 is an eight-bit counter with a carry-out signal. A GAL16V8 or PAL16R8 device can be used to build an eight-bit counter. However to provide a carry-out and/or carry-in signal, more outputs are required. The GAL18V10 fits the bill nicely, since it is a functional superset of the already flexible GAL16V8. Adding a few extra lines of equations in the source file and re-compiling produces a JEDEC file for a totally pin-compatible replacement, but with extra functionality. Other uses for the additional output macrocells include implementing nine or ten bit counters and decoded outputs from eight or nine bit counters. All of these functions could be done in a 22V10 as well (at extra cost), but could not be done in any of the common 20-pin PAL devices. Below is an example of two implementations of an eight-bit counter with carry-in and carry-out. While this design fits in one GAL18V10, it would require two different 20-pin PAL devices.

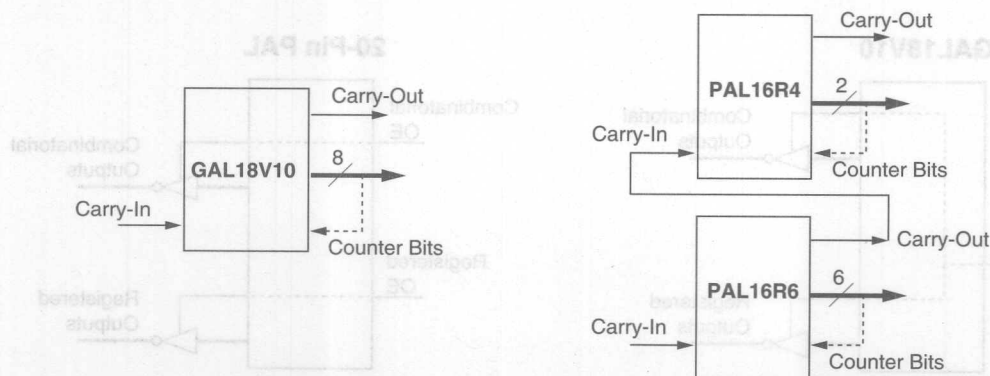


Figure 1. 8-bit Counter with Carry-In and Carry-Out

The GAL18V10 Advantage

Reset and Preset

Another benefit of the 22V10 nature of the GAL18V10 is the inclusion of Asynchronous Reset and Synchronous Preset of the registers. These dedicated product terms can allow any pin or combination of inputs and/or feedbacks to trigger a global reset or preset to occur. In many other devices this can only be accomplished by using valuable product terms and extra design time to build this capability into the logic for each output. Since each output in these devices has only seven or eight product terms, the addition of the reset/preset logic may make it impossible to fit in the desired logic functions. Example 1 illustrates what the eighth output of an eight-bit counter may look like.

To add the capability for a synchronous preset would require the use of an additional product term, which may not be available. This same problem may come up in a complex state machine.

The Asynchronous Reset function cannot even be duplicated in the GAL16V8 or standard PAL devices. The GAL18V10 can be asynchronously reset, therefore simplifying the power-up routine by not requiring a clock cycle to put the device into a known state.

Flexible Output Enable

Again because of its exact emulation of the common 20-pin PAL devices, the GAL16V8 has limited options for placement of the Output Enable control pins. A GAL16V8 with any I/O macrocells configured in registered mode always has pin 11 dedicated to the output enable of the register. Pin 11 is then no longer available as an input to the array. This means that any combinatorial outputs that need output enable control must use an additional pin, since the output enable control of combinatorial outputs is through a product term. A design with a mix of registered and combinatorial outputs using a GAL16V8 (or 20-pin PAL device) must always use two pins to get

```

Q7 := ( Q0 & Q1 & Q2 & Q3 & Q4 & Q5 & Q6 & !Q7      "Product Term 1
#      !Q6 & Q7      "Product Term 2
#      !Q5 & Q7      "Product Term 3
#      !Q4 & Q7      "Product Term 4
#      !Q3 & Q7      "Product Term 5
#      !Q2 & Q7      "Product Term 6
#      !Q1 & Q7      "Product Term 7
#      !Q0 & Q7 );   "Product Term 8

#      !Synch_Preset " NO MORE PRODUCT TERMS AVAILABLE!

```

Example 1. Eighth Output of an Eight-bit Counter

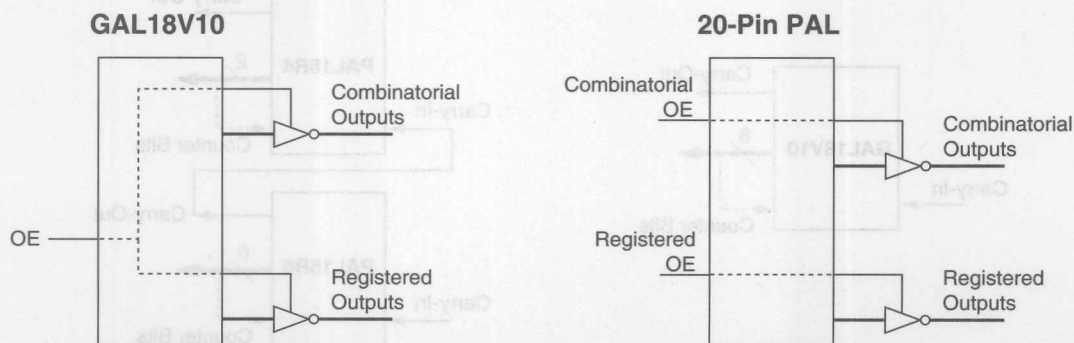


Figure 2. Output Enable Pin Consolidation

output enable control on all outputs. The GAL18V10 has no such restrictions. All output enable control is from a product term, regardless of whether the output is configured as registered or combinatorial.

Saving one pin on a 20-pin device can mean the difference between keeping the design in a 20-pin device and having to go to a larger (and more expensive) device. Figure 2 illustrates how the GAL18V10 can use one less pin than a GAL16V8 or 20-pin PAL device when both registered and combinatorial outputs must be tri-stated.

Conclusion

It is clear that standard PAL architectures have definite limitations. Lattice first addressed this issue with the GAL16V8 and GAL20V8 devices, which were able to replace all standard 20 and 24-pin PAL devices. For replacing those same PAL devices, and adding some additional flexibility, the GAL16V8 and GAL20V8 devices are a vast improvement and have become an industry standard in their own right.

However, as the previous examples have pointed out, there are many cases where the old standby programmable logic architectures, and even the first-generation GAL replacements, don't have the flexibility required. For 20-pin devices, the GAL18V10 provides complete design flexibility by using the familiar 22V10 architecture, while maintaining the ability to provide a pin-compatible superset of the GAL16V8.

output enable control on all outputs. The GAL18V10 has no such restrictions. All output enable control is from a product term, regardless of whether the output is configured as registered or combinational.

Saving one pin on a 20-pin device can mean the difference between keeping the design in a 20-pin device and having to go to a larger (and more expensive) device. Figure 2 illustrates how the GAL18V10 can use one less pin than a GAL16V8 or 20-pin PAL device when both registered and combinational outputs must be installed.

Combinational Outputs

It is clear that standard PAL architectures have definite limitations. Lattice first addressed this issue with the GAL16V8 and GAL20V8 devices, which were able to replace all standard 20 and 24-pin PAL devices. For replacing those same PAL devices, and adding some additional flexibility, the GAL18V8 and GAL20V8 devices are a vast improvement and have become an industry standard in their own right.

However, as the previous examples have pointed out, there are many cases where the old standby programmable logic architectures, and even the first-generation GAL replacements, don't have the flexibility required. For 20-pin devices, the GAL18V10 provides complete design flexibility by using the familiar 22V10 architecture, while maintaining the ability to provide a pin-compatible subset of the GAL16V8.



GAL20RA10: Programmable Clocks Improve System Performance

Introduction

There is a growing need for innovative design techniques to increase the system throughput using the currently available Programmable Logic Devices (PLDs). One way of improving the system throughput is to make use of Lattice's GAL20RA10. By taking advantage of the unique architecture featuring an individually controlled clock on each of the Output Logic Macro Cell (OLMC) registers, the resolution of the control signals generated by a GAL20RA10-based state machine can be doubled. The design example shown in this Application Brief takes advantage of this feature in a Dynamic RAM (DRAM) control logic design.

Design Example

The most common control signals generated by DRAM control logic are the Row Address Strobe (RAS) and the Column Address Strobe (CAS). The timing requirements of these control signals are strictly governed by the DRAM's timing requirements. Based on the DRAM's

timing requirement, Figure 1a shows how the RAS and CAS control signals are generated from a standard PLD device which has only one dedicated active high clock signal driving all the output registers. The basic constraint of the high-to-low transition of RAS signal to the high-to-low transition of CAS signal for the 100ns DRAM is 15ns minimum (row address hold time after RAS).

As illustrated in Figure 1b, the activation of CAS signal is unnecessarily delayed because of the limitations of the standard PLD's single active-high clock driving all the output registers. This limitation is not a reflection of the DRAM's requirements but rather a limitation of the standard PLD.

Design engineers can improve this design method by using the GAL20RA10's independent OLMC clocks. Figure 2b shows the RAS and CAS signals being driven by both the rising and falling edges of the clock signal. This technique can be implemented in the GAL20RA10 by simply feeding the complement of the clock input from the RAS control register to the CAS control register, as shown in figure 2a.

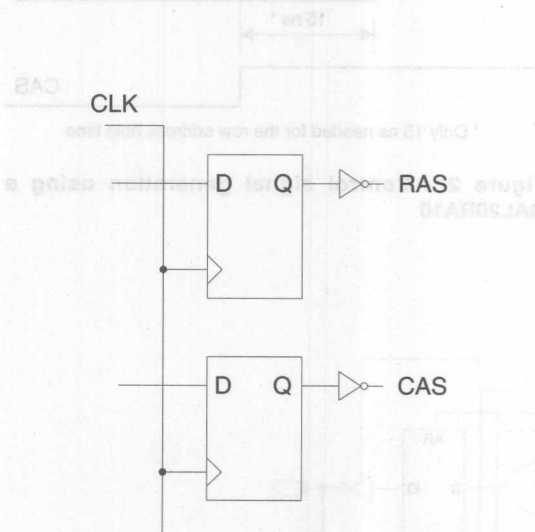


Figure 1a. Standard PLD design with a single active-high clock

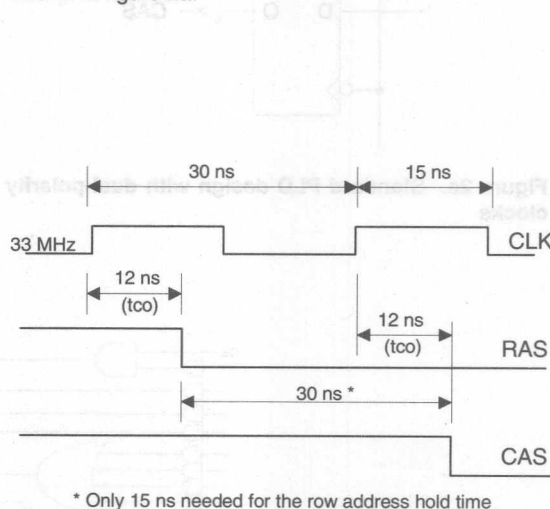


Figure 1b. Control signal generation with a single active-high clock

Programmable Clocks Improve System Performance

Summary

The individual clock control on the OLMC is one of several different product-term controlled features which are available on the GAL20RA10. Other individual product-term controlled signals of the GAL20RA10 include the Asynchronous Preset (AP) and Asynchronous Reset (AR). These signals can also be used, similar to the clock signal, to enhance system performance. In addition to these features, the GAL20RA10 has an external preload (PL) capability to improve the control over the register contents — especially in state machine design. The full GAL20RA10 macrocell architecture is shown below, in figure 3.

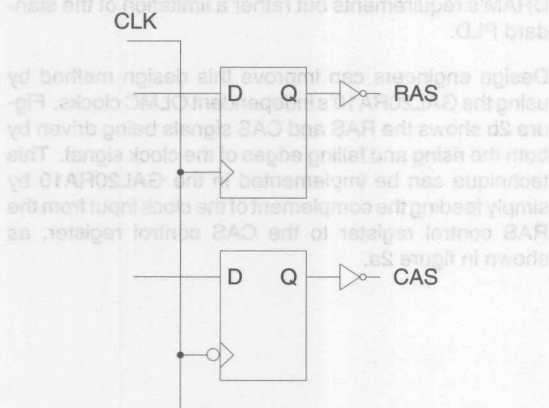


Figure 2a. Standard PLD design with dual-polarity clocks

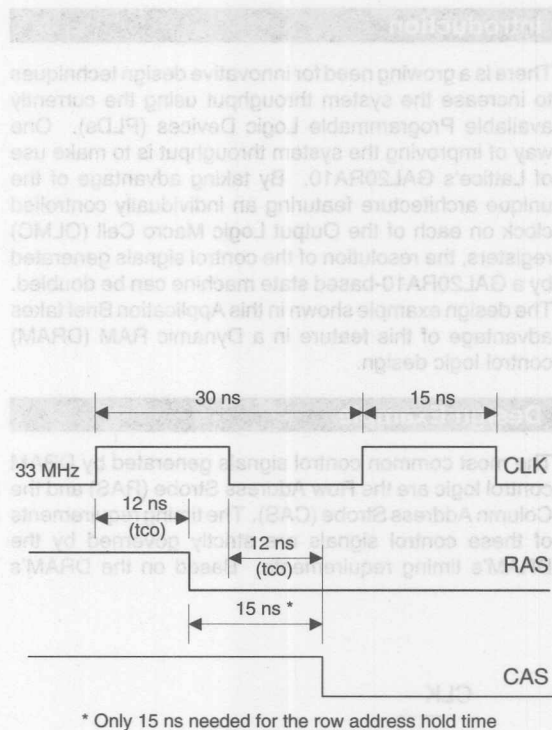


Figure 2b. Control signal generation using a GAL20RA10

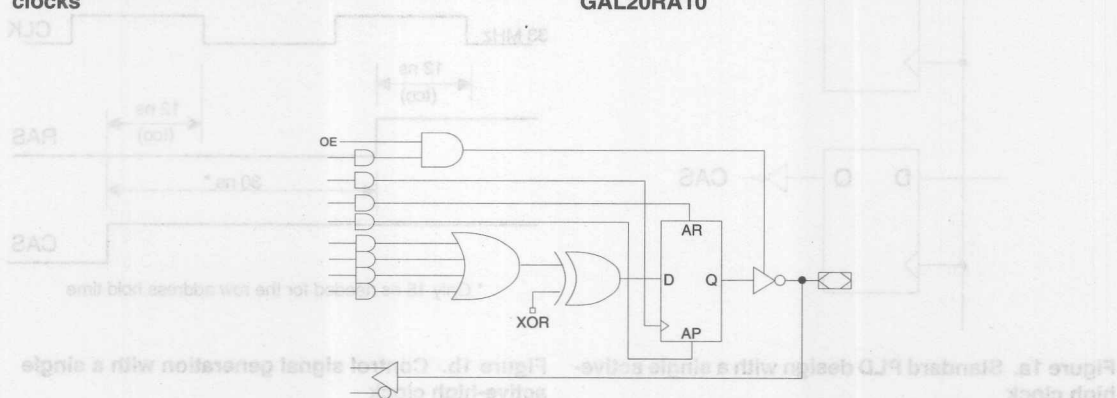


Figure 3. GAL20RA10 Macrocell



GAL6002 Designs Using ABEL and CUPL

Introduction

The Lattice GAL6002 is the most dense 24 pin PLD available. The GAL6002 is an FPLA (Field Programmable Logic Array) which has a programmable AND array and a programmable OR array. It has a total of thirty eight registers and two synchronous clock pins. Each of the logic outputs can be configured for multiple clock control.

This document covers the ABEL and CUPL syntax required to fully utilize the GAL6002 and its many features. Contained in this application note is an example source file for both the ABEL and CUPL compilers that has been compiled. This utilizes the features described in this application note.

Architecture Description

The GAL6002 has an FPLA architecture that contains both a programmable AND array and a programmable OR array. Inputs from the I/O pins as well as from internal registered feedback signals are brought into the AND array. The AND array then feeds the OR array, which in turn feeds the logic macrocells.

All input signals from the outside world that come into the AND array may be registered, latched, or directly connected to the AND array. Access from the outside world into the AND array is obtained through the Input Logic Macrocell (ILMC) or by the Input Output Logic Macrocell (IOLMC). Latch or register control is applied through a pin called I/CLK. The I/CLK pin may also be brought into the AND array and used as normal input. Inputs into the AND array from the Output Logic Macrocells (OLMCs) as well as from the Buried Logic Macrocells (BLMCs) are fed directly to the AND array.

The architecture of the GAL6002 allows the OLMC to be used as a BLMC if the inverting output buffer is placed into the high impedance state. The non-driven I/Os by way of the IOLMC then have access to the AND array where they may be used.

Logic function formation is done in either the OLMC or the BLMC. There are ten OLMCs and eight BLMCs. Only the ten OLMCs have access to pins and the AND array, while the BLMCs are fed back to the AND array. The OLMC has

a polarity selection capability at the D input of the Flip Flop, where as the BLMC does not.

The output of the OLMC is passed to the pins through an inverting buffer. The buffer controls output enable by using a product term.

BLMC / OLMC Configurations

Each of the OLMCs as well as the BLMCs have three possible configurations, listed below for the ABEL syntax.

1. Combinational
2. DE - type register, synchronously clocked, with a clock enable
3. D - type register, synchronously clocked

The OLMC and BLMC, if configured for registered operation, share a product term that is used for resetting the D-Flip-Flop. Each OLMC and BLMC in the registered mode of operation may be clocked from the dedicated clock pin called OCLK. If OCLK is used, then the clock may be gated off with a Clock Enable Sum Of Terms which may be unique to each OLMC or BLMC. Otherwise, a unique clock may be created from a Sum Of Terms. If a Clock Enable or a Clock Sum Of Terms is used, then either a positive or negative edge may be selected as the active edge.

ABEL Syntax and Logic Construction

Node Definitions Specific to ABEL for the use of pins as inputs to be either latched or registered.

Pn#	Fnt.	Nd.	Pn#	Fnt.	Nd.
"1	Clk		24	VCC	
"2	InR	70	23	InR	I/O 89
"3	InR	71	22	InR	I/O 88
"4	InR	72	21	InR	I/O 87
"5	InR	73	20	InR	I/O 86
"6	InR	74	19	InR	I/O 85
"7	InR	75	18	InR	I/O 84
"8	InR	76	17	InR	I/O 83
"9	InR	77	16	InR	I/O 82
"10	InR	78	15	InR	I/O 81
"11	InR	79	14	InR	I/O 80
"12	GND		13	CLK	

GAL6002 Designs Using ABEL and CUPL

Buried register node definitions:

nodes 33, 32, 31, 30, 29, 28, 27, 26;

Device Declaration:

XXXX device 'f6002';

Input Pin declared as combinatorial:

No declaration statements are required as this is the default condition of the inputs.

Input Pin declared as registered:

IR0 node 75; "Present in the declaration section of source file

Pin7 Pin 7;

EQUATIONS

IR0.D = Pin7; "Present in the declaration section of source file

IR0.C = Pin1;

Input Pin declared as latched:

IL0 node 70; "This line should be present prior to the equations statement

EQUATIONS

IL0.D = Pin2; "Defining inputs to the latches

IL0.LE = Pin1; "Defining Latch control to the latched inputs

I/O Pin declared as combinatorial input:

No declaration statements are required as this is the default condition of the inputs.

I/O Pin used as input and declared as registered:

IR0_I/O node 80; "Present in the declaration section of source file

Pin14 Pin 14;

EQUATIONS

IR0_I/O .D = Pin14; "Present in the declaration section of source file

IR0_I/O .C = Pin1;

I/O Pin used as input and declared as latched:

IL0_I/O node 81; "This line should be present prior to the equations statement

EQUATIONS

IL0_I/O .D = Pin15; "Defining inputs to the latches

IL0_I/O .LE = Pin1; "Defining Latch control to the latched inputs

OLMC as a combinatorial:

"I/O pins definitions

Pin23 Pin 23;

Pin23 istype 'com';

EQUATIONS

Pin23 = IL0.Q & IR0.Q;

"Straight combinatorial feed through of signals that have been previously latched and registered

OLMC as D-type flip-flop:

Pin21 Pin 21;

Pin21 istype 'reg_d'; "Clocking by product term.

EQUATIONS

Pin21 := IL2.Q & IR2.Q;

"Pin21 is registering values that have been latched and registered at the input pins. Default is clocking through OCLK pin.

OLMC as D-type flip-flop with a gated clock:

Pin22 Pin 22;

Pin22 istype 'reg_g'; "clocking by OCLK .CE control

EQUATIONS

Pin22 := IL1.Q & IR1.Q; "Pin22 is registering values that have been latched and registered at the input pins

Pin22.ce = IL3.Q; "Clock enable control

is by a latched value at the input Pins Clocking of Pin22 is through the clock pin OCLK

BLMC as a combinatorial:

BN7 node 33;
BN7 istype 'com'; "This definition may be omitted as it is achieved by default

EQUATIONS
BN7= Pin10& IL0.Q & IR0.Q;

BLMC as D-type flip-flop:

BN7 node 33;
BN7 istype 'reg_d';

EQUATIONS
BN7:= Pin10;
BN7.clk = IR0.Q; "Clock input may come from OCLK or from the array Default clocking is through OCLK pin.

BLMC as D-type flip-flop with a gated clock:

BN6 node 32;
BN6 istype 'reg_g'; "clocking by OCLK .CE control

EQUATIONS
BN6 := IL1.Q & IR1.Q; "Equation is clocked by the pin OCLK
BN6.ce = IL3.Q; "Clock enable is by use of a latched input

CUPL Syntax and Logic Construction

Buried register node definitions:

Nodes 25 through to 32 are assigned to BLMCs 0 through 7 respectively.

OLMC used as Buried Logic Macrocells

Use of the OLMC as buried registers requires the use of node numbers and the NODE declaration. The node numbers 33 through 42 are assigned to the OLMCs 14 through 23 respectively. If the OLMC are not to be buried then the PIN declaration statement is to be used.

Device Declaration:

Device g6002;

Inputs declared as combinatorial:

Inputs that do not have extensions associated with them are configured as combinatorial inputs that go straight through to the AND array.

Example of usage—

Pin declarations:
Pin 2 = IN2;
Pin 14 = OUT;

Use in equations:
OUT = IN2;

Use of I/O pins as combinatorial inputs—

Pin declarations:
Pin 15 = OUT2;

Use in equations:
OUT = OUT2;

Inputs Declared as registered:

The extension .DQ declares pins that are used as inputs to be configured as registered.

Example of usage—

Use of dedicated input pins as registered inputs—

Pin declarations:
Pin 14 = OUT;
Pin 2 = IN2;

Use in equations:
OUT = IN2.DQ;

Use of I/O pins as registered inputs to combinatorial outputs—

Pin declarations:
Pin 14 = OUT;
Pin 15 = OUT2;

Use in equations:
OUT = OUT2.DQ;

GAL6002 Designs Using ABEL and CUPL

Inputs Declared as latched:

The extension .IOL declares pins that are used as inputs to be configured as latched.

Example of usage—

Use of dedicated input pins as latched inputs—

Pin declarations:

```
Pin 14 = OUT;
```

```
Pin 2 = IN2;
```

Use in equations:

```
OUT = IN2.IOL;
```

Use of I/O pins as latched inputs—

Pin declarations:

```
Pin 14 = OUT;
```

```
Pin 15 = OUT2;
```

Use in equations:

```
OUT = OUT2.IOL;
```

BLMC & OLMC as a combinatorial:

For use as a combinatorial output no extension is to be used as this is the default state of the outputs.

Example of OLMC usage—

Pin declarations:

```
Pin 2 = IN2;
```

```
Pin 15 = OUT2;
```

Use in equations:

```
OUT2 = OUT2;
```

```
OUT2.OE = IN2;
```

If an output enable equation is not present the compiler causes the output always to be driven..

Example of BLMC usage—

Pin declaration and BLMC declaration:

```
Pin 2 = IN2;
```

```
NODE BLMCOUT1;
```

Use in equations:

```
BLMCOUT1 = IN2;
```

Note that there is no output enable capability in the BLMC as there is no direct access to the pins. Outputs from the BLMC are brought directly back into the AND array.

BLMC & OLMC as D-type flip-flop:

Clock for this configuration is obtained from a product term rather than the dedicated clock pin.

Example of OLMC usage—

Pin declarations:

```
Pin 2 = IN2;
```

```
Pin 15 = OUT2;
```

Use in equations:

```
OUT2.CK = IN2;
```

```
OUT2.OE = IN2;
```

```
OUT2.AR = IN2;
```

```
OUT2.D = OUT2 & IN2;
```

Example of BLMC usage—

Pin declaration and BLMC declaration:

```
Pin 2 = IN2;
```

```
NODE BLMCOUT1;
```

Use in equations:

```
BLMCOUT1.CK = IN2;
```

```
BLMCOUT1.AR = IN2;
```

```
BLMCOUT1.D = OUT2 & IN2;
```

BLMC & OLMC as D-type flip-flop with a gated Zclock:

Clock for this configuration is obtained from the dedicated clock pin rather than a product term. The clock can be gated off by use of a Clock Enable product term.

Example of OLMC usage—

Pin declarations and BLMC declaration:

```
Pin 2 = IN2;
```

```
Pin 15 = OUT2;
```

Use in equations:

```
OUT2.CE = IN2;
```

```
OUT2.OE = IN2;
```

```
OUT2.AR = IN2;
```

```
OUT2.D = OUT2 & IN2;
```

Example of BLMC usage—

Pin declarations:

```
Pin 2 = IN2;
```

```
NODE BLMCOUT1;
```

Use in equations:

```
BLMCOUT1.CE = IN2;
```

```
BLMCOUT1.AR = IN2;
```

```
BLMCOUT1.D = OUT2 & IN2;
```


Introduction

The GAL6002 is the most versatile 24-pin PLD available today. Its FPLA architecture offers buried macrocells, D/E registers, programmable clocks and dedicated input pins which can be individually configured as latches or registers. These features combine to provide the designer with an ideal platform on which to build complex state machines and other complex logic functions.

This application note will provide an example of how a GAL6002 might be used in a system. Additionally, it will show how software tools are used to exploit some of the device's unique features. The circuit to be described is a 4-to-1 RS-232 serial port multiplexer (Port MUX). The concept for the circuit arose from the need to replace a mechanical switch used to connect four computers to a high speed laser printer.

The Port MUX application uses every input and output pin, as well as all eight State Logic Macrocells. Other than a single GAL6002, the only ICs needed are RS-232 line driver/receiver chips and a clock source.

Basic RS-232 Protocol

To understand the operation of the port MUX, it is necessary to have a basic knowledge of RS-232 communications protocol.

Though the RS-232 protocol is standardized, its definition is loose enough to allow liberties to be taken in its implementation. When the port MUX was designed, the assumption was made that communication can take place with only four signals: transmit data, receive data, printer ready/busy, and computer ready/busy. In RS-232 parlance these signals are called TxD, RxD, DTR, and DSR, respectively.

An RS-232 link is digital (bistable) in nature, but the voltages used to represent logic ones and zeros are not TTL level. Instead, -12VDC represents a logic one and +12VDC represents a logic zero. Another consideration is that the idle or deasserted state of an RS-232 signal is a logic one, although data is transmitted in its "true" form. A typical single byte transfer can be seen in figure 1.

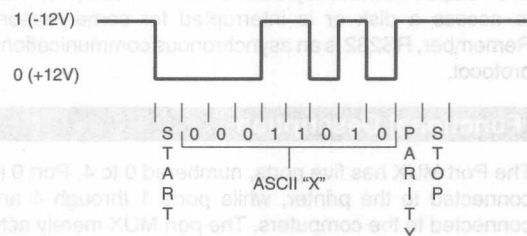


Figure 1. TxD during single byte transfer

A typical RS-232 data transfer between a computer and a printer would proceed as follows:

1) The printer, being powered-up and ready to accept data, has its DTR line asserted (logic zero), while its TxD line is idle (logic one). The computer, also powered-up but not yet sending data, is in a similar state: TxD is idle and DSR is asserted.

2) When the computer is ready to send a byte of data, it asserts its TxD line for 1 "bit period." This is called the start bit. A "bit period" is dependent on the data transmission speed (300 baud (bits/sec), 9600 baud, etc.). After the start bit, 7 or 8 bits of data will follow, optionally followed by a parity bit, and ending with 1 or 2 stop bits (logic one). Data is transmitted least significant bit first.

Note: Asserted = 0 (+12V)

"Idle" = 1 (-12V)

The condition of TxD and DSR after sending a byte of data is the same as before sending it. So, from an electrical perspective, there is no indication whether or not the computer is going to send another byte. If it is going to, it simply does so "when it feels like it."

3) Somewhere in the middle of the transfer, the printer runs out of paper, or its print buffer fills up, or for some other reason it must suspend communications. When this happens, the printer deasserts its DTR line, telling the computer to stop sending data. When the printer is again ready to accept data, it will reassert DTR.

As alluded to in #2 above, there is no absolute way to tell when the computer is finished sending data. In fact, the computer can be said to have "finished" its transmission

GAL6002: 4-to-1 RS232 Port Multiplexer

after sending only the first byte of a multi-byte transmission. Each subsequent byte transfer can be viewed as an entirely new transaction. Extended periods of time may even elapse between byte transfers if the computer has to access a disk or is interrupted for some reason. Remember, RS232 is an asynchronous communications protocol.

Functional Description

The Port MUX has five ports, numbered 0 to 4. Port 0 is connected to the printer, while ports 1 through 4 are connected to the computers. The port MUX merely acts as an intelligent switch; data flows through it unhindered and unaltered. At any given time, there will always be one (and only one) computer connected to the printer.

Four signals per port are switched: TxD, RxD, DTR, and DSR. This arrangement is known to work for connecting IBM-PC compatible computers to an HP LaserJet. The RS-232 specification has no lack of ready/busy signals, so others could be substituted for DTR and DSR if necessary (CTS and RTS, for example). See figure 2 for a block diagram of the port multiplexer.

Since RS-232 signal levels are not compatible with TTL levels, line driver/receiver circuits are needed for translation. For this project, Maxim MAX235 Driver/Receiver chips were used, though others devices will work as well. Each MAX235 IC has five drivers and five receivers; two MAX235 ICs are needed to build the Port MUX.

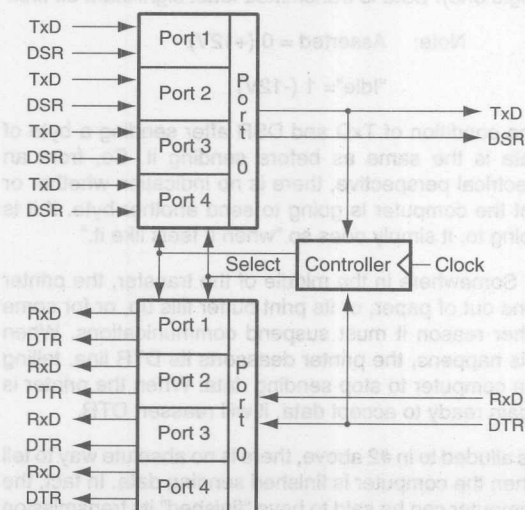


Figure 2. Block diagram of Port MUX

The multiplexer functions by sequentially scanning the four input ports until data appears at one of them.

Scanning a port involves connecting that port to the printer and waiting for data to flow. If no data appears within a predetermined time period, the period of the system clock (~.25s), the process is repeated at the next port. When data does appear at a port, the port MUX "locks onto" that port and goes into the transmit mode. At the end of the transmission, the port MUX returns to the scan mode.

As mentioned in the discussion of RS-232 protocol, detecting the end of a transmission is non-trivial. To peripheral devices such as printers, the "end of transmission" concept is fiction — to them, life is one big data transmission. The port MUX, on the other hand, must be able to determine when it is permissible to resume scanning. It should not return to the scan mode before the end of a transmission, and at the same time it must not lock onto a port for an inordinately long time. Both requirements are met by timing how long the computer's TxD line is idle, and returning to the scan mode if TxD is idle for longer than a predetermined time period (5 - 10 seconds is reasonable).

The data routing logic of the port multiplexer is controlled by two loosely coupled state machines and a status register. The state machines and the status register use the State Logic Macrocells (labeled as state bits S0 - S7). The status register determines the operating mode (scan or transmit); the first state machine determines the active port; and the second state machine is used as a timer.

The basis for most state machines is the simple binary counter, with added logic to allow branching, state skipping, etc. The most efficient way to build a binary counter in the GAL6002 is to configure the registers to emulate T-flip flops. This way, only the conditions that should cause the state bits to change state need to be specified. In the case of simple up counters, there is only one condition when all lower order bits are ones. The equations for a 4-bit up counter are as follows:

$$\begin{aligned} B0.D &= /B0.Q; \\ B0.E &= 1; \end{aligned}$$

$$\begin{aligned} B1.D &= /B1.Q; \\ B1.E &= B0; \end{aligned}$$

$$\begin{aligned} B2.D &= /B2.Q; \\ B2.E &= B1*B0; \end{aligned}$$

$$\begin{aligned} B3.D &= /B3.Q; \\ B3.E &= B2*B1*B0; \end{aligned}$$

As you can see, counters of any size can be built using only two product terms per bit.

GAL6002: 4-to-1 RS232 Port Multiplexer

In the following discussions POTx and PODT are TxD and DTR respectively.

Status Register

Recall that the beginning of a data transfer is signaled by POTx becoming active for one bit period. By using the start bit event to asynchronously set a status register, set = transmit, the operating mode of the port MUX is determined. Once set, the status bit will remain set until a time-out occurs.

The status register is implemented using state bit S0, configured to emulate a T-flip flop with a programmable clock. With such an arrangement, meeting the specified state transition conditions doesn't just allow a transition at the next clock, but actually causes the transition.

There are two situations that must cause the status register to toggle: if it is clear, clear = scan, and data is flowing, or if it is set and a timeout has occurred. The equations for the status register are:

$$\begin{aligned} S0.D &= /SO.Q; \\ S0.CK &= /SO.Q*/POTx + \\ &\quad SO.Q*POTx*PODT*57.Q \\ &\quad *56.Q*55.Q*54.Q*53.Q; \end{aligned}$$

The same function could have been implemented by "building" a latch from combinational equations, but the approach taken here is more efficient in terms of product term usage and is less prone to functional hazards.

Primary State Machine

The primary state machine directly determines the active port. It is simply a 2-bit counter with a hold function. The conditions necessary for the counter to increment are that the status register be clear and that PODT be active.

The primary state machine uses state bits S1 and S2 in the D/E configuration to emulate T-flip flops. The equations for S1 and S2 are:

$$\begin{aligned} S1.D &= /S1.Q; \\ S1.E &= PODT*/SO.Q; \\ S2.D &= /S2.Q; \\ S2.E &= PODT*/SO.Q*S1.Q; \end{aligned}$$

Timer

The second state machine, a 5-bit counter/timer, will only count while the status register is set, POTx is idle, and PODT is active. The counter synchronously resets to zero if these conditions are not met. Thus, if PODT is

active and POTx is idle on 31 consecutive OCLK edges, the timer will reach its maximum value, causing the status register to be cleared and the primary state machine to continue counting.

The 5-bit timer uses state bits S4 - S7 in the D/E configuration, again emulating T-flip flops. The equations for the timer are:

$$\begin{aligned} S3.D &= /S3.Q*SO.Q*POTx*PODT; \\ S3.E &= 1; \\ S4.D &= /S4.Q*SO.Q*POTx*PODT; \\ S4.E &= S3.Q \\ &\quad + /SO.Q; \\ S5.D &= /S5.Q*SO.Q*POTx*PODT; \\ S5.E &= S3.Q*S4.Q \\ &\quad + /SO.Q; \\ S6.D &= /S6.Q*SO.Q*POTx*PODT; \\ S6.E &= S3.Q*S4.Q*S5.Q \\ &\quad + /SO.Q; \\ S7.D &= /S7.Q*SO.Q*POTx*PODT; \\ S7.E &= S3.Q*S4.Q*S5.Q*S6.Q \\ &\quad + /SO.Q; \end{aligned}$$

Time-out during a byte transfer, though statistically possible, is unlikely. If it should occur, however, it is not harmful. Because a time-out simply clears the status register and scanning does not resume until the next OCLK edge, the driving computer still has one OCLK period to finish the byte transfer (plenty of time!), during which time a logic zero on POTx returns the status register to the transmit mode. Thus, it is virtually impossible for a byte of data to be lost.

If the port MUX should time-out and resume scanning between byte transfers, but before the end of a transmission, and another computer is waiting to use the printer, then that computer will be serviced before the first computer is again granted use of the printer. This would cause the second computer's data to be inserted into the middle of the first computer's transmission. This situation, though undesirable, is unavoidable. The good news is that the probability of this happening is very low.

Conclusion

The Port MUX provides a "real life" example of how the flexibility of the GAL6002 can simplify a complex design. Equally important, this example shows how various software tools are used to take advantage of the device's features. Unfortunately, the Port MUX is not a speed critical application; in fact, the GAL6002's 15ns t_{PD} is overkill.

GAL6002: 4-to-1 RS232 Port Multiplexer

Though this example is complex, it still does not push the GAL6002 to its limits. The state machine and data routing equations use only 33 product terms, leaving over 48% of the AND array free for expansion. The GAL6002's FPLA architecture allowed 5 product terms to be merged. If this design were implemented using a standard 24-pin PLD, it would take at least two devices to accomplish this task.

In the following discussions POTX and POUT are TXD and DTR respectively.

Status Register

Recall that the beginning of a data transfer is signaled by POTX becoming active for one bit period. By using the start bit even to asynchronously set a status register, set = transmit, the operating mode of the port MUX is determined. Once set, the status bit will remain set until a time-out occurs.

The status register is implemented using state bit S0, configured to emulate a T-flip flop with a programmable clock. With such an arrangement, meeting the specified state transition conditions doesn't just allow a transition at the next clock, but actually causes the transition.

There are two situations that must cause the status register to toggle: if it is clear, clear = scan, and data is flowing, or if it is set and a timeout has occurred. The equations for the status register are:

$$\begin{aligned} S0.D &= \text{clear} + \text{clear} + \text{clear} + \text{clear} \\ S0.CK &= \text{clear} + \text{clear} + \text{clear} + \text{clear} \end{aligned}$$

The same function could have been implemented by "building" a latch from combinational equations, but the approach taken here is more efficient in terms of product term usage and is less prone to functional hazards.

Primary State Machine

The primary state machine directly determines the active port. It is simply a 2-bit counter with a hold function. The conditions necessary for the counter to increment are that the status register be clear and that POUT be active.

The primary state machine uses state bits S1 and S2 in the OE configuration to emulate T-flip flops. The equations for S1 and S2 are:

$$\begin{aligned} S1.D &= \text{clear} + \text{clear} + \text{clear} + \text{clear} \\ S1.CK &= \text{clear} + \text{clear} + \text{clear} + \text{clear} \\ S2.D &= \text{clear} + \text{clear} + \text{clear} + \text{clear} \\ S2.CK &= \text{clear} + \text{clear} + \text{clear} + \text{clear} \end{aligned}$$

Secondary State Machine

The second state machine, a 2-bit counter, will only count while the status register is set, POTX is false, and POUT is active. The counter synchronously resets to zero if these conditions are not met. Thus, if POUT is

Time-out during a byte transfer, through statistically possible, is unlikely. If it should occur, however, it is not a problem. Because a time-out simply clears the status register and scanning does not resume until the next CLK edge, the driving computer still has one CLK period to finish the byte transfer (plenty of time), during which time a logic zero on POTX returns the status register to the transmit mode. Thus, it is virtually impossible for a byte of data to be lost.

If the port MUX should time-out and resume scanning between byte transfers, but before the end of a transmission, and another computer is waiting to use the printer, then that computer will be serviced before the first computer is again granted use of the printer. This would cause the second computer's data to be inserted into the middle of the first computer's transmission. This situation, though undesirable, is unavoidable. The good news is that the probability of this happening is very low.

Conclusion

The Port MUX provides a "real life" example of how the flexibility of the GAL6002 can simplify a complex design. Equally important, this example shows how various software tools are used to take advantage of the device's features. Unfortunately, the Port MUX is not a speed critical application. In fact, the GAL6002's 12ns tag would

Introduction

The GAL22V10 provides a quick solution to bus arbitration and control needs. In this application note, we discuss how a VME bus arbitration circuit can be easily implemented within a GAL22V10, while leaving logic available on the GAL for other functions.

In any bus-oriented system, there may be multiple devices that need exclusive access to the bus. You need to provide some form of arbitration, ensuring that only one device has control of the bus at any point in time. This arbitration function is an ideal candidate for a GAL-based solution. For this example, we have chosen the VME bus standard and describe an approach for a GAL22V10 VME bus arbiter, including the ABEL source code for programming.

Design Example

A VME-based system is a good example of a bus that supports multiple bus masters, by requiring bus arbitration logic on the bus to resolve possible conflicting requests. The VME bus has a bus request line that a device asserts when it wants to gain control of the bus. The arbiter prioritizes these bus requests and asserts a bus grant to the device with the highest priority. Since you may wish to give some devices higher priority to bus access than others, the VME bus provides four bus request lines, with BR3 designated as the highest priority request, and BR0 as the lowest priority request. And since you may have more than four devices on the bus, or you may want to have more than one device sharing the same priority level, the bus standard must also support multiple masters with the same priority level. The VME bus allows more than one device on the bus to share a bus request line by "daisy-chaining" the bus grant signal — the device physically closest to the bus arbitrator "sees" the bus grant first. If this device didn't generate the bus request, then the bus grant is allowed to proceed to the next device on the bus. Figure 1 shows a simplified block diagram of a VME system, showing the handshaking between the arbiter and the other devices on the bus.

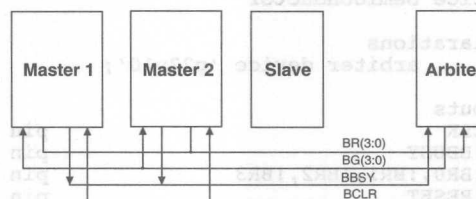


Figure 1. The Bus Arbitration Process

The bus arbitration process is defined in the following steps (the *level* of a bus master indicates which bus request line it used to gain access to the bus) :

- 1) A bus request signal (BR3:0) is received by the arbiter.
- 2) If the bus is not busy, the arbiter returns a corresponding bus grant (BG3:0)
- 3) If the bus is busy, the arbiter asserts a bus clear request (BCLR) as long as one of the following conditions is true:

- The current bus master is level 2, 1, or 0, and the active bus request line is 3 or 2.
- The current bus master is level 0, and any other device is requesting the bus

This protocol allows a higher priority device to take control of the bus in an orderly manner — the arbiter asserts BCLR, the current bus master releases BBSY, and then the higher priority device gains control of the bus. Also note that if the current bus master is 3, then the arbiter gives control to another level 3 requester only when BBSY goes inactive — in other words, a level 3 master will be allowed uninterrupted access to the bus.

ABEL Source

The following is an ABEL source file that implements a VME bus arbiter in a GAL22V10. Note that the internal signals, Master1 and Master2, are used to hold the level of the bus request currently being served.

Title
 Bus Arbiter with Priority Handling for GAL22V10
 Lattice Semiconductor'

Declarations

arbiter device 'p22v10';

"Inputs

CLK pin 1;
 !BBUSY pin 2;
 !BR0,!BR1,!BR2,!BR3 pin 3,4,5,6;
 !RESET pin 11;
 !OE pin 13;

"Outputs

!BCLR pin 23 istype 'invert,reg_d';
 !BGIN0,!BGIN1,!BGIN2,!BGIN3 pin 22,21,20,19 istype 'invert,reg_d';
 MASTER1,MASTER0 pin 15,14 istype 'buffer,reg_d';

MASTER = [MASTER1, MASTER0]; C, X, Z, L, H = .C., .X., .Z., 0, 1;

Equations

BGIN3 := !BBUSY & BR3
 # !BBUSY & BGIN3;
 BGIN3.C = CLK;
 BGIN3.AR = RESET;
 BGIN3.OE = OE;
 BGIN2 := !BBUSY & !BR3 & BR2
 # !BBUSY & BGIN2;
 BGIN2.C = CLK;
 BGIN2.AR = RESET;
 BGIN2.OE = OE;
 BGIN1 := !BBUSY & !BR3 & !BR2 & BR1
 # !BBUSY & BGIN1;
 BGIN1.C = CLK;
 BGIN1.AR = RESET;
 BGIN1.OE = OE;
 BGIN0 := !BBUSY & !BR3 & !BR2 & !BR1 & BR0
 # !BBUSY & BGIN0;
 BGIN0.C = CLK;
 BGIN0.AR = RESET;
 BGIN0.OE = OE;
 MASTER1 := BBUSY & BGIN3 "Master MSB
 # BBUSY & BGIN2
 # BBUSY & MASTER1;
 MASTER0 := BBUSY & BGIN3 "Master LSB
 # BBUSY & BGIN1
 # BBUSY & MASTER0;
 MASTER.AR = RESET;
 MASTER.OE = OE;
 MASTER.C = CLK;
 BCLR := BBUSY & BR3 & (MASTER <= 2)
 # BBUSY & BR2 & (MASTER <= 2)
 # BBUSY & BR1 & (MASTER <= 1)
 # BBUSY & BR0 & (MASTER == 0)
 # BBUSY & BCLR;
 BCLR.AR = RESET;

VME Bus Arbitration Using a GAL22V10

```
BCLR.OE      = OE;
BCLR.C       = CLK;
```

Test_vectors

```
([CLK,!RESET,!BBUSY,!BR3,!BR2,!BR1,!BR0,!OE]->
[!BCLR,!BGIN3,!BGIN2,!BGIN1,!BGIN0,MASTER])
```

```
"      !!                      ! ! ! ! M
"      R B                      ! B B B B A
"      E B ! ! ! !              B G G G G S
"      C S U B B B B !          C I I I I T
"      L E S R R R R O          L N N N N E
"      L T Y 3 2 1 0 E          R 3 2 1 0 R
"
```

```
[X,X,X,X,X,X,X,1]->[Z,Z,Z,Z,Z,Z,Z];"tristate
[C,1,1,1,1,1,0,0]->[H,H,H,H,L,0];"BR0 request
[C,1,1,1,1,1,0,0]->[H,H,H,H,L,0];
[C,1,1,1,1,1,0,0]->[H,H,H,H,L,0];
[C,1,0,1,1,1,1,0]->[H,H,H,H,H,0];
[C,1,0,1,1,1,1,0]->[H,H,H,H,H,0];
[C,1,0,1,1,1,0,0]->[L,H,H,H,H,0];"test bus clear line >= 0
[C,1,1,1,1,0,0,0]->[H,H,H,L,H,0];"BR1 request higher priority
[C,1,0,1,1,1,1,0]->[H,H,H,H,H,1];
[C,1,0,1,1,1,0,0]->[H,H,H,H,H,1];"test bus clear line >= 1
[C,1,0,1,1,0,1,0]->[L,H,H,H,H,1];
```

```
[C,1,1,1,0,0,0,0]->[H,H,L,H,H,0];
[C,1,0,1,1,1,1,0]->[H,H,H,H,H,2];
[C,1,0,1,1,1,0,0]->[H,H,H,H,H,2];
[C,1,0,1,1,0,1,0]->[H,H,H,H,H,2];
[C,1,0,1,0,1,1,0]->[L,H,H,H,H,2];
```

"BR2 request higher priority

"test bus clear line >= 2

```
[C,1,1,0,1,1,1,0]->[H,L,H,H,H,0];
[C,1,0,1,1,1,1,0]->[H,H,H,H,H,3];
[C,1,0,1,1,1,0,0]->[H,H,H,H,H,3];
[C,1,0,1,1,0,1,0]->[H,H,H,H,H,3];
[C,1,0,1,0,1,1,0]->[H,H,H,H,H,3];
[C,1,0,0,1,1,1,0]->[H,H,H,H,H,3];
[X,0,X,X,X,X,X,0]->[H,H,H,H,H,0];
```

"BR3 request higher priority

"test bus clear line
"requests ignored

"reset

END

Notes

BCLR.C = CLR;
BCLR.ON = OR;

Test vectors
[CLR, RESET, BR3, BR2, BR1, BR0, JOE] ->
[BCLR, BGIN3, BGIN2, BGIN1, BGIN0, MASTER]

```

"      1 1 1 1 1 1
"      R B 1 1 1 1
"      C S U B B B B 1
"      T E S T V E C T O R S
"      T Y 3 3 1 0 R

```

```

[X,X,X,X,X,X,X,X,1]->[X,X,X,X,X,X,X,X,1]; "crstace
[C,1,1,1,1,0,0,0]->[H,H,H,H,L,0]; "BR0 request
[C,1,1,1,1,0,0,0]->[H,H,H,H,L,0];
[C,1,1,1,1,0,0,0]->[H,H,H,H,L,0];
[C,1,1,1,1,0,0,0]->[H,H,H,H,L,0];
[C,1,0,1,1,1,1,0,0]->[H,H,H,H,H,0];
[C,1,0,1,1,1,1,0,0]->[H,H,H,H,H,0];
[C,1,0,1,1,1,1,0,0]->[H,H,H,H,H,0]; "test bus clear line >= 0
[C,1,0,1,1,1,0,0,0]->[H,H,H,H,L,0]; "BR1 request higher priority
[C,1,1,1,1,0,0,0,0]->[H,H,H,H,L,0];
[C,1,0,1,1,1,1,0,0]->[H,H,H,H,H,1];
[C,1,0,1,1,1,1,0,0]->[H,H,H,H,H,1]; "test bus clear line >= 1
[C,1,0,1,1,0,1,0,0]->[H,H,H,H,H,1];
[C,1,0,1,1,0,1,0,0]->[L,H,H,H,H,1];

```

```

[C,1,1,1,0,0,0,0,0]->[H,H,L,H,H,0]; "BR2 request higher priority
[C,1,0,1,1,1,1,0,0]->[H,H,H,H,H,2];
[C,1,0,1,1,1,0,0,0]->[H,H,H,H,H,2];
[C,1,0,1,1,0,1,0,0]->[H,H,H,H,H,2];
[C,1,0,1,0,1,1,0,0]->[L,H,H,H,H,2];
[C,1,1,0,1,1,1,1,0,0]->[H,L,H,H,H,0];
[C,1,0,1,1,1,1,1,0,0]->[H,H,H,H,H,3];
[C,1,0,1,1,1,0,0,0]->[H,H,H,H,H,3];
[C,1,0,1,1,0,1,0,0]->[H,H,H,H,H,3];
[C,1,0,1,0,1,1,0,0]->[H,H,H,H,H,3];
[C,1,0,0,1,1,1,0,0]->[H,H,H,H,H,3];
[C,1,0,0,1,1,1,0,0]->[H,H,H,H,H,3];
[X,0,X,X,X,X,X,0]->[H,H,H,H,H,0];

```

END



GAL16VP8/20VP8: Bus Arbitration Circuit

Introduction

Lattice's GAL16VP8 and GAL20VP8 devices combine the programmable logic flexibility of the industry standard GAL16V8 and GAL20V8 devices with the high output drive capability of the 74240 series of TTL devices. The GAL16VP8 and GAL20VP8 devices have been designed to implement bus and memory interface logic as a one chip solution, as opposed to the historical two chip solution.

The addition of 64mA output drive capability, and the option to individually configure the outputs either as standard totem pole outputs or open-drain outputs, has taken these 16/20VP8 devices beyond the realm of common glue logic integration. These capabilities allow the devices to drive heavy capacitive loads in applications such as bus and memory address and control signal drivers.

The following bus arbitration design example takes full advantage of the bus driving capability and the open-drain option of the GAL16VP8 to implement the bus arbitration circuit.

Design Example

A bus arbitration circuit is used to determine which board connected to the system bus gets control of the bus for data transfers. One of the most common methods of arbitrating the bus is to assign a priority level to each board and to award bus ownership to the board with the highest priority request. This normally requires a combination of a priority encoder and decoder logic to determine the priority. The GAL16VP8 highlighted in this design example uses wired-OR logic on the bus, implemented with open-drain outputs, to determine the relative priority, eliminating the use of dedicated priority encoder/decoder logic. This scheme is similar to the one used in IBM's Micro Channel bus standard.

Using this scheme, the board with the lowest numeric value ID has the highest priority — 0000 being the highest priority and 1111 being the lowest priority. Priority is resolved between competing boards by making the arbitration outputs (ARB3-ARB0) and bus request signals (BREQ) open-drain with external pull-up resistors.

A typical bus arbitration cycle begins by detection of an active BREQ signal driven by all requesting boards. (If BREQ is inactive, it means that none of the boards on the bus has requested the bus.) The requesting boards drive the ARB3-ARB0 signals according to the predetermined priority that is assigned by the ID3-ID0 inputs. The bit-by-bit resolution of the arbitration begins with a comparison at ARB3. When comparing the ARB3 bit, a logic low on ARB3 will prevail over a logic high on ARB3 since the bus has a wired-OR structure.

The comparison continues with ARB2, which compares ARB3 from the previous stage as well as the prevailing ARB2. The requesting boards that do not match the priority driven on ARB3 will no longer drive ARB2-ARB0. For example, if ARB3=0 on the bus but a board has ID3=1, the board will no longer drive the low order bits (ARB2-ARB0). This process continues until ARB0 is reached. After resolving ARB0, the lowest numerical value (highest priority) will be driven on ARB3-ARB0. A local bus grant signal is generated from the result of the arbitration if the value on ARB3-ARB0 matches that on the ID3-ID0 inputs to a given board. The board winning the bus keeps BREQ active until its bus access is complete.

An example CUPL source file which implements the arbitration logic is shown on the following pages. Figure 1 illustrates a typical bus interface block diagram. As a footnote to the open-drain configuration, use the FUSES statement and the fuse numbers provided in the datasheet to implement open-drain outputs in ABEL.

GAL16VP8/20VP8: Bus Arbitration Circuit

Summary

The unique ability for a logic device to drive 64mA and the option to have open-drain outputs have made the GAL16VP8 and GAL20VP8 ideal for bus and memory interface logic. The GAL16/20VP8 devices allow the design engineer a one chip solution for complex memory or bus applications that require 64mA IOL. The one chip solution reduces power and cost, while increasing speed and reliability.

A typical bus arbitration cycle begins by detection of an active BREQ signal driven by all requesting boards. If BREQ is inactive, it means that none of the boards on the bus has requested the bus. The requesting boards drive the ARB0-ARB3 signals according to the predetermined priority that is assigned by the ID0-ID3 inputs. The bit-by-bit resolution of the arbitration begins with a comparison at ARB0. When comparing the ARB0 bit, a logic low on ARB0 will prevail over a logic high on ARB0 since the bus has a wired-OR structure.

The comparison continues with ARB1, which compares ARB1 from the previous stage as well as the prevailing ARB0. The requesting boards that do not match the ARB0-ARB1 will no longer drive ARB2-ARB3. For example, if ARB0=0 on the bus but a board has ID3=1, the board will no longer drive the low order bits (ARB2-ARB0). This process continues until ARB3 is reached. After resolving ARB0, the lowest numerical value (highest priority) will be driven on ARB3-ARB0. A local bus grant signal is generated from the result of the arbitration if the value on ARB3-ARB0 matches that on the ID3-ID0 inputs to a given board. The board winning the bus keeps BREQ active until its bus access is complete.

An example CUPF source file which implements the arbitration logic is shown on the following page. Figure 1 illustrates a typical bus interface block diagram. As a footnote to the open-drain configuration, use the FUSE3 statement and the fuse numbers provided in the datasheet to implement open-drain outputs in ABEL.

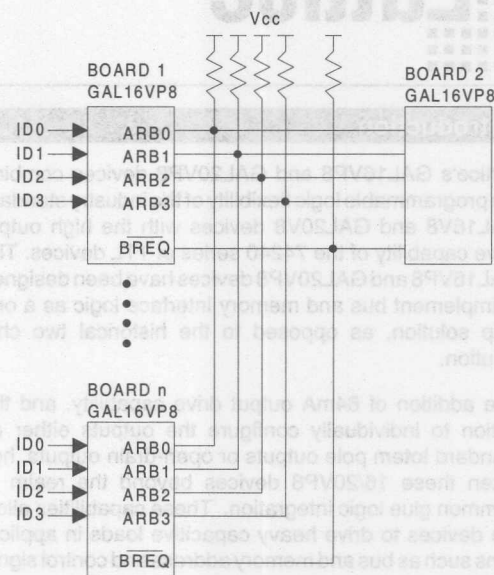


Figure 1. Typical System Interface.

GAL16VP8/20VP8: Bus Arbitration Circuit

Listing 1. Example of a CUPL Source File.

```
Name          VP8ARB;
Partno        U00;
Device        G16VP8MA;
Revision      1.0;
Date          09/09/99;
Designer      John Doe;
Company       Lattice Semiconductor Corp.;
Location      Hillsboro, OR;
Assembly      16VP8 bus arbitration circuit;

/*****
/* This bus arbitration circuit is based on the priority
/* assigned to the board. ID3..0 is configured to the board
/* priority code. This example uses ID 3..1=0111 for the
/* priority code. ARB3..0 is driven to the same priority
/* code.
*****/

/** Dedicated Inputs definition**/

Pin 1 = ID3; /* priority ID must be driven to .... */
Pin 2 = ID2; /* ... the appropriate priority levels ...*/
Pin 3 = ID1; /* ... by the local board.
Pin 4 = ID0;

/** Note Pin 5 is Vcc on this package **/
Pin 6 = LARB; /* Local arbitrate signal. Enables local board arbitration */

/** I/O pin definitions **/

Pin 17 = BREQ; /* active low open-drain bus request */
Pin 12 = ARB3; /* open-drain arbitration bits */
```

GAL16VP8/20VP8: Bus Arbitration Circuit

```

Pin 13 = ARB2;
Pin 14 = ARB1;
Pin 16 = ARB0;

/** Note Pin 15 is GND on this package **/

/** Output Equations **/

!BREQ = !ARB3 & ARB2 & ARB1 & ARB0;
BREQ.TEC = 'b'0; /* .TEC=0 specifies the open-drain output..TEC=1 or
                  the default is the totem pole output. */
BREQ.OE = LARB; /* Enables local board arbitration */

ARB3 = !ARB3 & !BREQ
      # ID3 & BREQ; /* arbitration bit 3 */
ARB3.TEC = 'b'0;
ARB3.OE = LARB; /* Enables local board arbitration */

ARB2 = !ARB3 & ARB2 & !BREQ
      # ID2 & BREQ; /* arbitration bit 2 */
ARB2.TEC = 'b'0;
ARB2.OE = LARB; /* Enables local board arbitration */

ARB1 = !ARB3 & ARB2 & ARB1 & !BREQ
      # ID1 & BREQ; /* arbitration bit 1 */
ARB1.TEC = 'b'0;
ARB1.OE = LARB; /* Enables local board arbitration */

ARB0 = !ARB3 & ARB2 & ARB1 & ARB0 & !BREQ
      # ID0 & BREQ; /* arbitration bit 0 */
ARB0.TEC = 'b'0;
ARB0.OE = LARB; /* Enables local board arbitration */

```



GAL20XV10: Data Block Transfer Address Detector

Introduction

The Exclusive-OR (XOR) gate can efficiently implement arithmetic functions such as counters, adders and decoders, using fewer product terms than the standard sum-of-products PLD architecture (AND, fixed OR array). This is demonstrated by logic equation example 1.

To take full advantage of product term usage in a high speed system design, a high speed device with a built-in XOR function is needed. The Lattice GAL20XV10 fills the need for such a device. The GAL20XV10 achieves a 10ns Tpd while consuming only 90mA Icc (Max.). The closest competitor's device offers only a Tpd of 30ns at 180mA Icc. In addition, the generic architecture of the GAL20XV10 gives system designers the ability to configure outputs to any combination of registers, combinatorial, XOR and AND/OR structures.

Design Example

An address counter that uses a comparator to keep track of the block data transfer is a typical application which illustrates the advantages of the GAL20XV10's XOR architecture. If the starting and ending addresses are given, the address counter will generate and increment

the transfer address. The comparator will then compare the counter bits with the ending address. When the counter value equals the ending address, the address comparator will issue a transfer complete signal. The following CUPL example source file (example 2) shows how this function can be implemented using CUPL compiler syntax. Notice that the syntax demonstrates the usage of .OE and .OEMUX to control the AND/OR product term configuration and XOR configuration, respectively.

Conclusion

This design example illustrates the efficient usage of the XOR function by implementing the address counter with 11 product terms instead of the 14 product terms required with a standard programmable AND, fixed OR configuration. The bit-wise comparator, implemented with the XOR function, also makes the design clear and understandable, as illustrated by the logic equations. With this efficient, easy to understand design, the system can run at up to 100MHz with 10ns tpd.

Example 1. XOR Logic Equation

\$ - XOR function syntax	& - AND function syntax
# - OR function syntax	! - INVERT function syntax
XOR Function	Equivalent AND/OR Function
A \$ B /* 2 PT used	(A & !B) # (!A & B) /* 2 PT used
(A & B) \$ (C & D) /* 2 PT used	(A & B & !C) # (A & B & !D) /* 4 PT used
	# (!A & C & D) # (!B & C & D)

GAL20XV10: Data Block Transfer Address Detector

Example 2. CUPL Source File

```
Name      APPXV10;
Partno    00;
Date      09/09/99;
Revision  00;
Designer  Jane Doe;
Company   Lattice;
Assembly  None;
Location  None;
Device    g20xv10;

/*****
/* This CUPL example uses the GAL20XV10 to build the
/* 4-bit up counter with load function and a 4-bit
/* comparator. This counter implementation takes
/* advantage of the built-in XOR function of the
/* GAL20XV10. It also shows the XOR and AND/OR
/* configuration in CUPL syntax
*****/

/** Input definition **/

PIN 1  = SYSCLK;
PIN 2  = SA0;      /* STARTING ADDRESS BITS */
PIN 3  = SA1;
PIN 4  = SA2;
PIN 5  = SA3;
PIN 6  = EA0;      /* ENDING ADDRESS BITS */
PIN 7  = EA1;
PIN 8  = EA2;
PIN 9  = EA3;
PIN 10 = STARTLD;  /* STARTING ADDRESS LOAD */
PIN 11 = OE_COMP;
PIN 13 = OUT_EN;

/** Output Definition **/

PIN 23 = !AC0;      /* ADDRESS COUNTER BITS */
PIN 22 = !AC1;
PIN 21 = !AC2;
PIN 20 = !AC3;
PIN 19 = !CMP0;      /* ADDRESS COMPARE BITS */
PIN 18 = !CMP1;
PIN 17 = !CMP2;
PIN 16 = !CMP3;
PIN 15 = EQUAL;      /* EQUALITY COMPARE */

/** Equations **/

AC0.D = !STARTLD & AC0      /** AC0 TOGGLE WITH CLOCK **/
      $ STARTLD & SA0;      /** LOAD SA0 **/
AC0.OEMUX = OUT_EN;
```

GAL20XV10: Data Block Transfer Address Detector

```
AC1.D = !STARTLD & AC0           /** AC1 CNT UP CONDITION **/  
      $ !STARTLD & AC1           /** TOGGLE AC1 **/  
      # STARTLD & SA1;           /** LOAD SA1 **/  
AC1.OEMUX = OUT_EN;  
  
AC2.D = !STARTLD & AC0 & AC1     /** AC2 CNT UP CONDITION **/  
      $ !STARTLD & AC2           /** TOGGLE AC2 **/  
      # STARTLD & SA2;           /** LOAD SA2 **/  
AC2.OEMUX = OUT_EN;  
  
AC3.D = !STARTLD & AC0 & AC1 & AC2 /** AC3 CNT UP CONDITION **/  
      $ !STARTLD & AC3           /** TOGGLE AC3 **/  
      # STARTLD & SA3;           /** LOAD SA3 **/  
AC3.OEMUX = OUT_EN;  
  
CMP0 = AC0 $ EA0;                /** COMPARE ADDR BIT0 **/  
CMP0.OEMUX = OUT_EN;  
  
CMP1 = AC1 $ EA1;                /** COMPARE ADDR BIT1 **/  
CMP1.OEMUX = OUT_EN;  
  
CMP2 = AC2 $ EA2;                /** COMPARE ADDR BIT2 **/  
CMP2.OEMUX = OUT_EN;  
  
CMP3 = AC3 $ EA3;                /** COMPARE ADDR BIT3 **/  
CMP3.OEMUX = OUT_EN;  
  
EQUAL = !CMP0 & !CMP1 & !CMP2 & !CMP3; /** MAGNITUDE COMPARE **/  
EQUAL.OE = OE_COMP;
```


Notes

```

ACI.D = 1STARTED & ACO
% 1STARTED & ACI
% 1STARTED & SA1
ACI.OMUX = OUT_EN;

ACI.CNT UP CONDITION **\
\ ** ACI CNT UP CONDITION **\
\ ** TOGGLE ACI **\
\ ** LOAD SA1 **\

AC2.D = 1STARTED & ACO & ACI
% 1STARTED & AC2
% 1STARTED & SA2
AC2.OMUX = OUT_EN;

AC2.CNT UP CONDITION **\
\ ** AC2 CNT UP CONDITION **\
\ ** TOGGLE AC2 **\
\ ** LOAD SA2 **\

AC3.D = 1STARTED & ACO & ACI & AC2
% 1STARTED & AC3
% 1STARTED & SA3
AC3.OMUX = OUT_EN;

CMP0 = ACO & SA0;
CMP0.OMUX = OUT_EN;

CMP1 = ACI & SA1;
CMP1.OMUX = OUT_EN;

CMP2 = AC2 & SA2;
CMP2.OMUX = OUT_EN;

CMP3 = AC3 & SA3;
CMP3.OMUX = OUT_EN;

EQUAL = 1CMP0 & 1CMP1 & 1CMP2 & 1CMP3;
EQUAL.OE = OE_COMP;

\ ** COMPARE ADDR BITS **\
\ ** COMPARE ADDR BITS **\
\ ** COMPARE ADDR BITS **\
\ ** COMPARE ADDR BITS **\
\ ** COMPARE ADDR BITS **\

```

Introduction

When designing with standard PLDs such as the GAL20V8 and GAL22V10, system design engineers are sometimes faced with a situation where a few extra product terms or extra macrocells are required to implement the design. These situations usually do not warrant adding a second standard PLD. The ideal solution is to find a way to add these extra product terms and/or outputs while still keeping the design in one device. The design example given in this applications brief illustrates one example of how the extra outputs of the GAL26CV12 can solve the common problem of needing additional outputs. The design will show a programmable frequency divider that uses a 10-bit counter as a base and can therefore divide the incoming frequency by up to 1024.

Design Example

The design requirements for the programmable logic device are 10 macrocells for the internal counter, one macrocell for the programmable output frequency, 4 inputs for the frequency selection and one input clock.

Figure 1 below shows the simple block diagram of the programmable frequency divider.

This frequency divider implementation, using D type registers, requires more than 8 product terms for the two most significant counter bits on the 10-bit counter. The programmable frequency output also requires more than 8 product terms. Therefore, even two GAL20V8 devices (or other standard PAL devices) would not work for this design, since they only have a maximum of 8 product terms per output. Since a total of 11 macrocells is required to implement the counter and the programmable frequency output, even a 22V10 device would not work.

A single GAL26CV12 device satisfies both the product term requirements and the output macrocell requirements for the example design. The equations and output pin assignments required to implement the 10-bit programmable frequency divider are provided in example 1. Notice that the outputs that require more than 8 product terms are assigned to the inner-most pins of the device, since the inner-most pins have the highest number of product terms available.

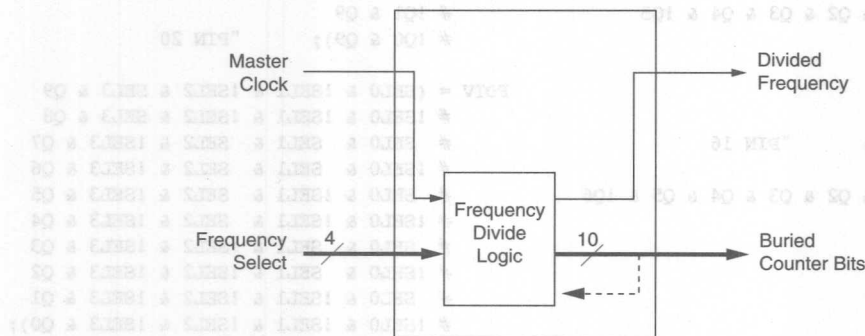


Figure 1. Block Diagram of Programmable Frequency Divider

GAL26CV12: Programmable Frequency Divider

Summary

The GAL26CV12 has a total of 12 output logic macrocells and a product term distribution of 8 terms on the outermost pins to 12 on the innermost pins. It comes in a 28-pin DIP and PLCC package, with center Vcc and Ground pins on the DIP package. When design engi-

neers are frustrated by the limitations on the number of available product terms, output macrocells, or input pins on standard PLD devices, using the GAL26CV12 is a valuable design alternative. Since the GAL26CV12 can often save the cost of adding a second PLD, the design is simplified while also cutting cost and board space requirements.

Example 1. Equations and Output Pin Assignments for a 10-Bit Programmable Frequency Divider

```

Q0.D = (!Q0);          "PIN 27
Q1.D = (Q0 & !Q1
# !Q0 & Q1);          "PIN 26
Q2.D = ( Q0 & Q1 & !Q2
# !Q1 & Q2
# !Q0 & Q2);          "PIN 25
Q3.D = (Q0 & Q1 & Q2 & !Q3
# !Q2 & Q3
# !Q1 & Q3
# !Q0 & Q3);          "PIN 23
Q4.D = (Q0 & Q1 & Q2 & Q3 & !Q4
# !Q3 & Q4
# !Q2 & Q4
# !Q1 & Q4
# !Q0 & Q4);          "PIN 15
Q5.D = (Q0 & Q1 & Q2 & Q3 & Q4 & !Q5
# !Q4 & Q5
# !Q3 & Q5
# !Q2 & Q5
# !Q1 & Q5
# !Q0 & Q5);          "PIN 16
Q6.D = (Q0 & Q1 & Q2 & Q3 & Q4 & Q5 & !Q6
# !Q5 & Q6
# !Q4 & Q6
# !Q3 & Q6
# !Q2 & Q6
# !Q1 & Q6
# !Q0 & Q6);          "PIN 17
Q7.D = (Q0 & Q1 & Q2 & Q3 & Q4 & Q5 & Q6
& !Q7
# !Q6 & Q7
# !Q5 & Q7
# !Q4 & Q7
# !Q3 & Q7
# !Q2 & Q7
# !Q1 & Q7
# !Q0 & Q7);          "PIN 18
Q8.D = (Q0 & Q1 & Q2 & Q3 & Q4 & Q5 & Q6 & Q7
& !Q8
# !Q7 & Q8
# !Q6 & Q8
# !Q5 & Q8
# !Q4 & Q8
# !Q3 & Q8
# !Q2 & Q8
# !Q1 & Q8
# !Q0 & Q8);          "PIN 19
Q9.D = (Q0 & Q1 & Q2 & Q3 & Q4 & Q5 & Q6 & Q7
& Q8 & !Q9
# !Q8 & Q9
# !Q7 & Q9
# !Q6 & Q9
# !Q5 & Q9
# !Q4 & Q9
# !Q3 & Q9
# !Q2 & Q9
# !Q1 & Q9
# !Q0 & Q9);          "PIN 20
FDIV = (SEL0 & !SEL1 & !SEL2 & SEL3 & Q9
# !SEL0 & !SEL1 & !SEL2 & SEL3 & Q8
# SEL0 & SEL1 & SEL2 & !SEL3 & Q7
# !SEL0 & SEL1 & SEL2 & !SEL3 & Q6
# SEL0 & !SEL1 & SEL2 & !SEL3 & Q5
# !SEL0 & !SEL1 & SEL2 & !SEL3 & Q4
# SEL0 & SEL1 & !SEL2 & !SEL3 & Q3
# !SEL0 & SEL1 & !SEL2 & !SEL3 & Q2
# SEL0 & !SEL1 & !SEL2 & !SEL3 & Q1
# !SEL0 & !SEL1 & !SEL2 & !SEL3 & Q0);
"PIN 22

```

Section 1: Introduction

Section 2: ispLSI and pLSI Architecture Overview

Section 3: ispLSI and pLSI Development Tools

Section 4: ispLSI and pLSI Application Notes

Section 5: GAL Architecture Overview

Section 6: GAL Development Tools

Section 7: GAL Application Notes

Section 8: In-System Programmable Generic Digital Switch (ispGDS)

Introduction to the ispGDS Family	8-1
Using ispGDS Devices	8-5
ispGDS Compiler Support	8-7
Lattice's Solution for Plug-and-Play	8-9

Section 9: Design Techniques

Section 10: Article Reprints

Section 11: Technology, Quality, and Reliability Overview

Section 12: General Section

Section 1: Introduction	
Section 2: iapLSI and qLSI Architecture Overview	
Section 3: iapLSI and qLSI Development Tools	
Section 4: iapLSI and qLSI Application Notes	
Section 5: GAL Architecture Overview	
Section 6: GAL Development Tools	
Section 7: GAL Application Notes	
Section 8: In-System Programmable General Digital Switch (ispGDS)	
Introduction to the ispGDS Family	8-1
Using ispGDS Devices	8-5
ispGDS Compiler Support	8-7
Lattice's Solution for Plug-and-Play	8-9
Section 9: Design Techniques	
Section 10: Article Reprints	
Section 11: Technology, Quality, and Reliability Overview	
Section 12: General Section	

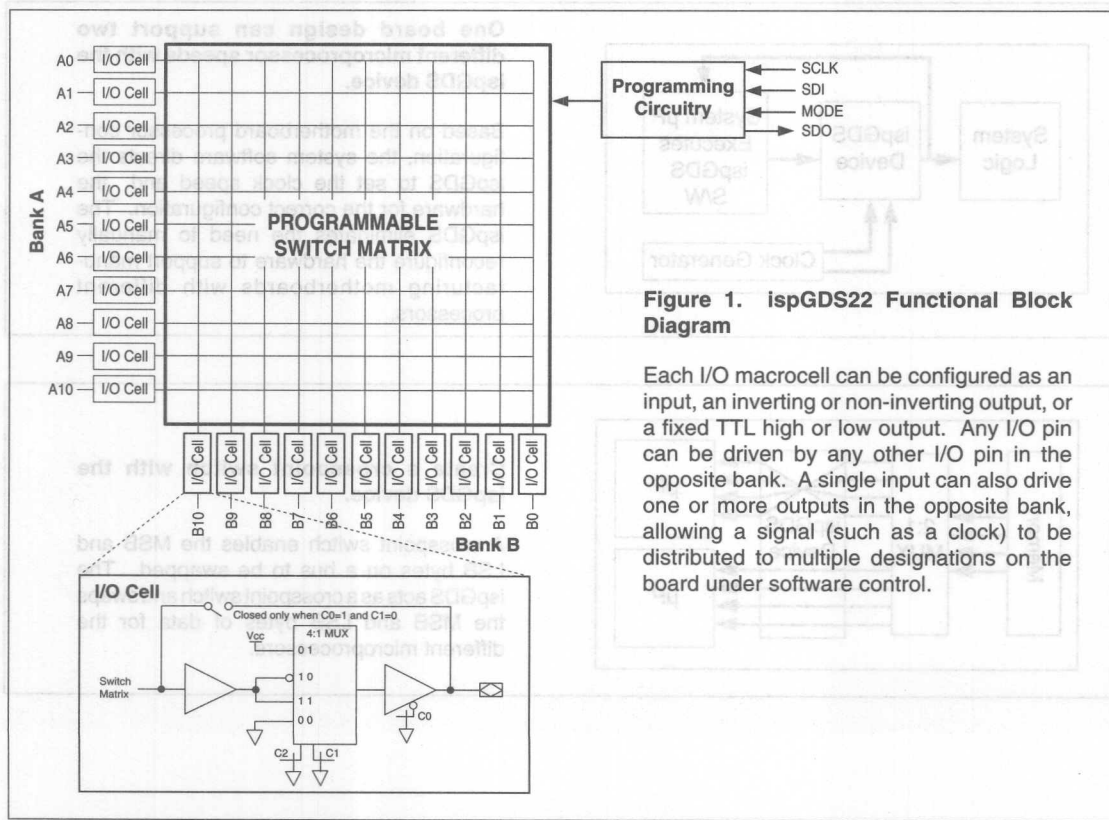
Introduction to the ispGDS Family

Lattice, the pioneer of non-volatile in-system programmable (ISP™) logic has expanded the application of ISP to include programmable system interconnect. The ispGDS (Generic Digital Switch) family combines the in-system programmability, high performance and low power of Lattice's GAL programmable logic technology with a switch matrix architecture, resulting in an innovative programmable signal router. The ispGDS device is a configurable switch matrix which provides the ability to quickly implement and change p.c. board connections without changing mechanical switches or other system hardware. ISP allows the connections to be reprogrammed without removal from the p.c. board via a simple 5V, 4-wire serial interface. This capability allows the system designer to define hardware which can be reconfigured in-system to meet a variety of applications. The ispGDS also conserves board real estate, providing up to 22 I/Os in about a quarter square inch of board space.

With today's demand for user-friendly systems, there is an increasing need for hardware which is easily reconfigured under software control without manual intervention. The Lattice ispGDS family is an ideal solution for end-system feature reconfiguration and signal routing applications. The fast 7.5ns propagation delay through the devices supports high-performance signal routing applications. Easier system upgrades, user feature selection and system manufacturing are the results.

The ispGDS device also provides higher quality and reliability than other switch solutions due to the nature of E²CMOS technology. E²CMOS technology supports 100% testability which guarantees 100% in-system programmability and functionality.

There are three members of the ispGDS family: the ispGDS22, ispGDS18, and ispGDS14. Each of the devices operates identically with the only difference being the number of I/O cells available.

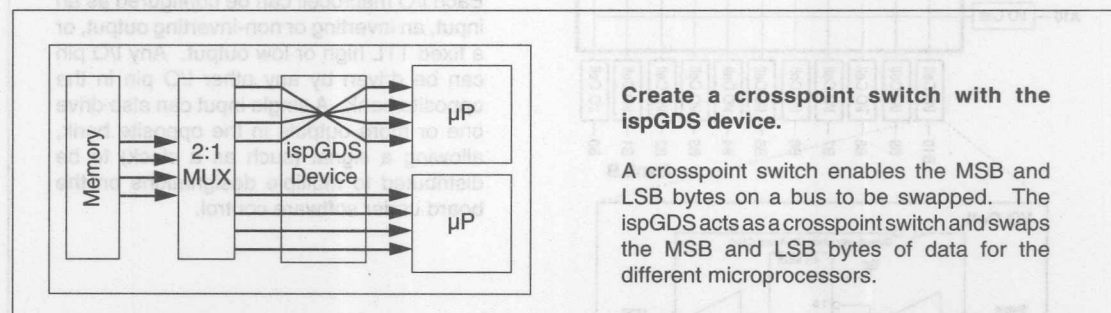
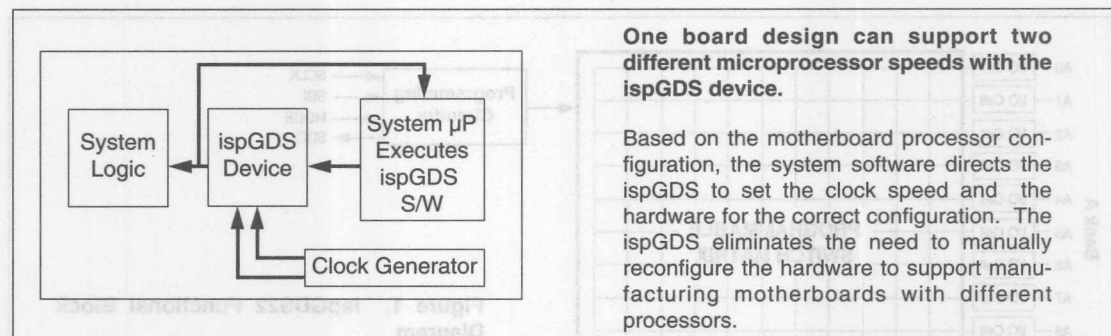
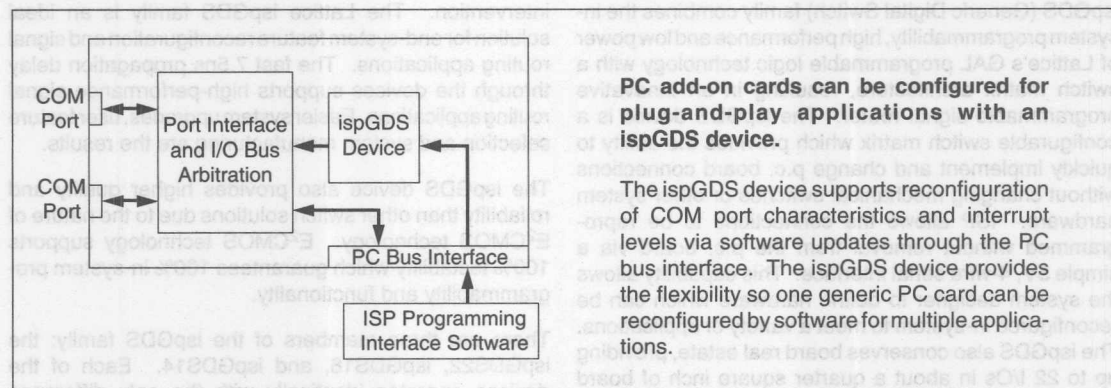


Introduction to ispGDS

ispGDS Applications

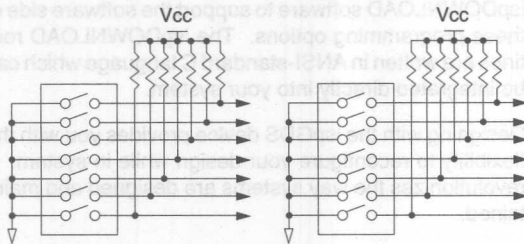
With ispGDS devices, designs can be reconfigured without mechanical devices or user intervention. Provision for easier system upgrades and feature selection can

now be included in the system's original design. A few examples of actual ispGDS applications demonstrate the possibilities.



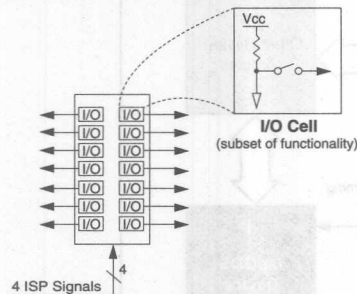
2 DIP Switches + 14 Resistors

14 pins which can be pulled high or low manually



ispGDS14

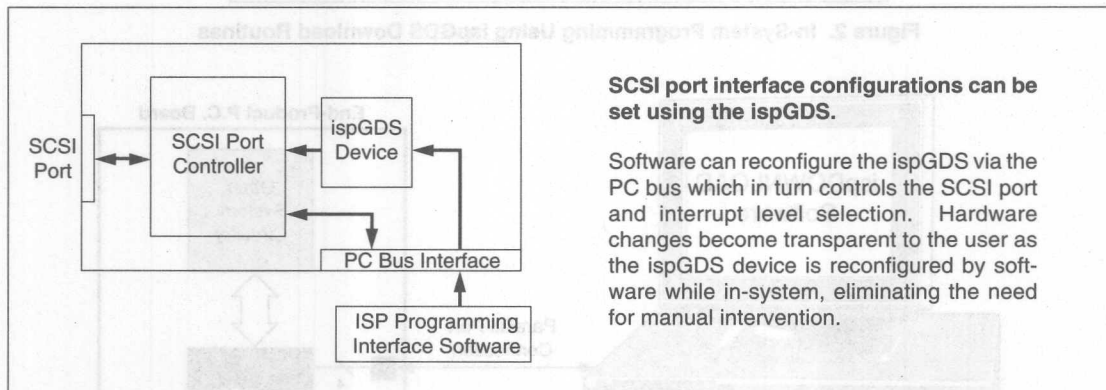
14 pins which can be set high or low by SOFTWARE



Replace DIP switches with a software controlled switch alternative.

The ispGDS can be configured as a programmable replacement for standard DIP switches, providing space savings, in-system reconfigurability, higher reliability as well as ease of use. The programmable nature of the ispGDS eliminates the need to manually select DIP switch settings.

8



SCSI port interface configurations can be set using the ispGDS.

Software can reconfigure the ispGDS via the PC bus which in turn controls the SCSI port and interrupt level selection. Hardware changes become transparent to the user as the ispGDS device is reconfigured by software while in-system, eliminating the need for manual intervention.

Introduction to ispGDS

In-System Programming

The ispGDS devices can be programmed in-system using 5 volt only signals through a simple 4-wire programming interface using TTL level signals. Programming and erasure of the entire device can be done in less than one second.

In addition to third party programmers, the ispGDS device can be programmed from your automatic test equipment (ATE) or even from a PC on your manufacturing line. For more flexibility, you can have your product's embedded microprocessor configure the ispGDS devices through one of its I/O ports, making a field upgrade a snap.

Lattice provides free compiler support and ispDOWNLOAD software to support the software side of these programming options. The ispDOWNLOAD routines are written in ANSI-standard C language which can be integrated directly into your system.

Designing with the ispGDS device provides you with the flexibility to reconfigure your design while in-system. It revolutionizes the way systems are designed and maintained.

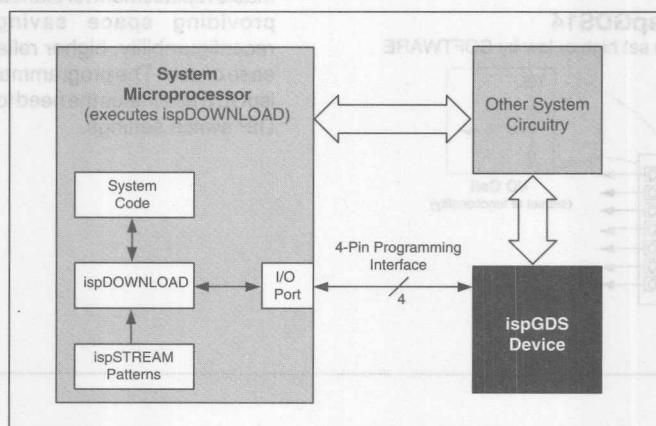


Figure 2. In-System Programming Using ispGDS Download Routines

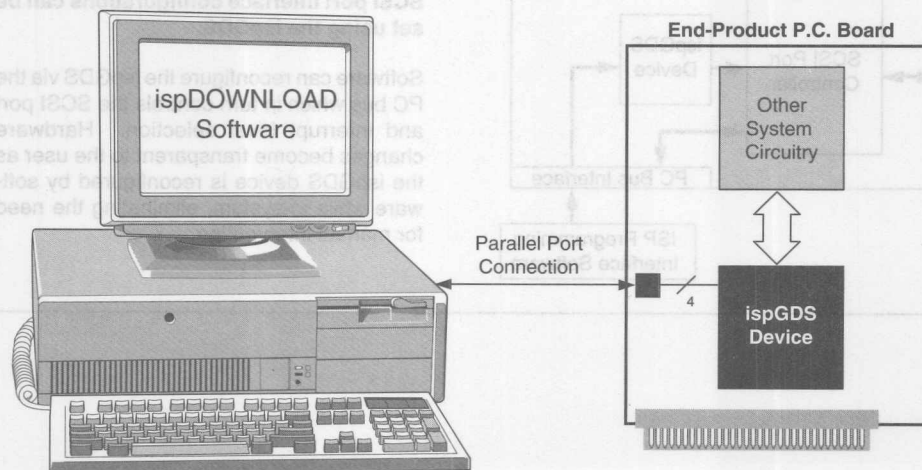


Figure 3. Configuring an ispGDS Device from a Remote System

Introduction

A common problem among board designs is that they require a means to configure hardware options, such as driving signals to a fixed high or low state, or controlling the routing of signals between two or more points. Although a DIP switch can solve these problems, it has the drawback that someone must physically set the switches, introducing the possibility of user error, mechanical damage, and the need for customer support to resolve these problems. The ispGDS family is a cost effective solution to these problems, since they can duplicate the functionality of a DIP switch without requiring manual switch setting. You also gain additional functionality through in-system programmability and nonvolatile E²CMOS storage of the switch configuration. By using the four-pin TTL interface for in-system programmability, you can configure the device under software control, allowing a user to change the hardware setup without physically removing a card or manipulating a DIP switch. By simplifying the task the user faces in configuring the hardware, you improve system reliability and ease of use while reducing your customer support requirements.

Significant Gains in Reliability and Cost

You can significantly improve system reliability using the ispGDS family. Since the ispGDS family can provide the same functionality with fewer pins than a DIP switch, and since ispGDS devices don't require pull-up resistors,

your design will require fewer solder joints and external parts, improving mechanical reliability and lowering assembly cost. The ispGDS family also provides additional capability for system test. Since you can set each I/O pin of an ispGDS device to either Vcc or Ground (GND), your system software can control the ispGDS device to provide test signals to other parts of the circuit, reducing test complexity and time.

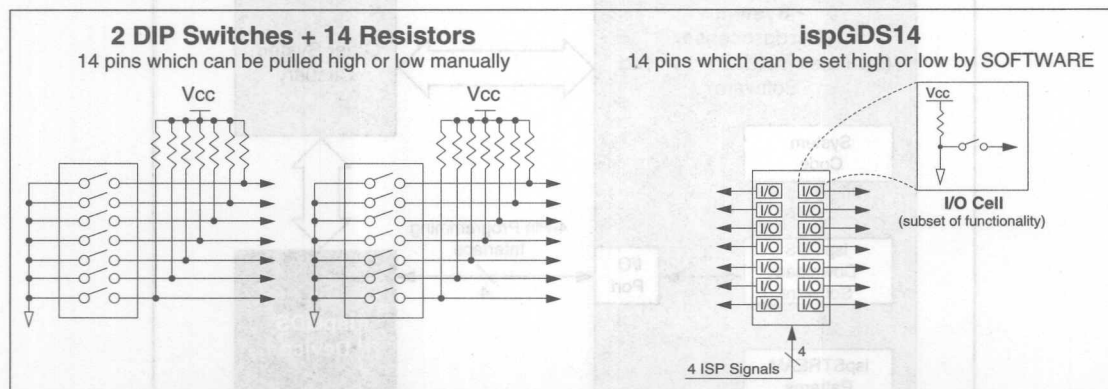
More Efficient Utilization of Board Space

In many applications, a DIP switch is configured with one side of the switch tied to GND, and the other side connected to pull-up resistors, as shown in the diagram below. The ispGDS family eliminates the need for these connections to GND and pullups, freeing package pins that would have been tied to GND to do something useful. For instance, an ispGDS14 has 14 pins available that can be internally tied to either Vcc or GND. This is accomplished in a 20 pin package. The equivalent DIP switch solution would require two 7-position DIP switches (which means two 14 pin packages) and pull-up resistors for each of the 14 switch outputs (see figure 1).

Digital Cross-Point Switch

Simple DIP switch replacement is not the only use for ispGDS devices. Since ispGDS devices are configured as two banks of I/O pins, with any pin in one bank able to make a connection to any pin in the other bank, an

Figure 1. ispGDS Devices Offer a Software Controlled Alternative to DIP Switches.



Using ispGDS Devices

ispGDS device can also function as a digital cross-point switch. This cross-point switch capability provides the designer with the ability to change the routing and distribution of signals under software control. For example, the ispGDS could select one of several interrupt lines from a bus and route to a single net on the board. Another example would be swapping MSB and LSB bytes from a databus.

A Variety of Different Matrix Sizes

Lattice offers the ispGDS in three different switch matrix sizes:

Device Name	Matrix Size	I/O Pins
ispGDS22	11 x 11	22
ispGDS18	9 x 9	18
ispGDS14	7 x 7	14

As noted above, the ispGDS22 provides an 11 x 11 cross-point matrix, the ispGDS18 provides a 9 x 9 matrix, and the ispGDS14 provides a 7 x 7 matrix. The size of the matrix indicates the size of the banks — for example, the ispGDS22 has two banks that are 11 pins wide. In this case, you can route any one of the 11 pins on one bank to any one, several or all of the 11 pins on the other bank.

Free ispGDS Programming Software

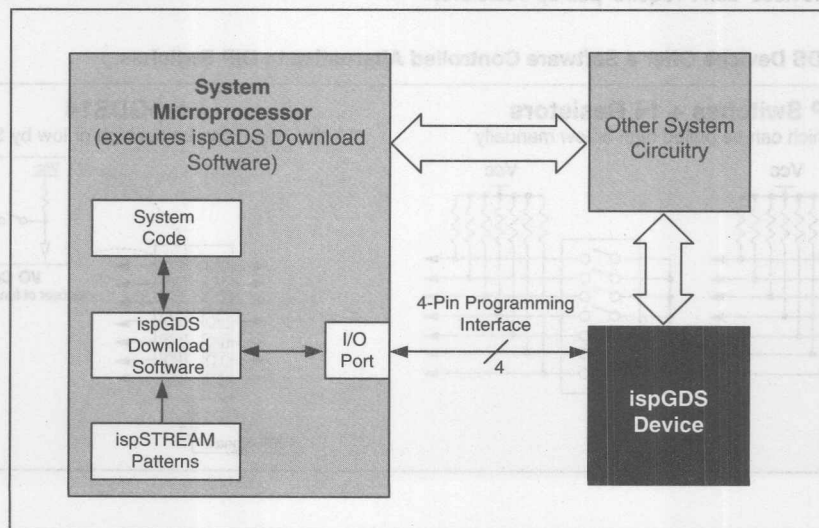
To assist you in designing software to program ispGDS devices, Lattice provides a library of ANSI-standard C language routines that implement the ispGDS programming algorithms. These routines allow you to program and read the devices simply by making a function call to the appropriate library function. Sample applications are provided which use the PC parallel port for programming, allowing you to program the ispGDS through a PC parallel port without modifying or compiling any code.

If you want to program ispGDS devices through some other custom interface, you can modify the ispGDS source code. The hardware dependent portion of the code is isolated in a few functions, allowing you to easily and quickly change this interface to accommodate your custom hardware needs. The diagram below shows a top level block diagram of a typical system using an ispGDS.

Summary

ispGDS devices offer DIP switch functionality and in-system programmability at a price that is competitive with traditional DIP switch approaches. You can also use an ispGDS device to emulate a digital cross-point switch, where any input on one bank can be driven to one or more outputs on any other bank. Through in-system programmability ispGDS devices can also provide software controlled test capability, by driving signals either high or low, or rerouting signals for test. Software controlled board-configuration is now a reality.

Figure 2. Typical System Using an ispGDS device and ispGDS Download Software



ispGDS Compiler Support

Introduction

To simplify the development of ispGDS devices, Lattice offers an ispGDS assembler named "GASM" which processes the input ASCII files to generate the JEDEC compatible fusemap files required for the ispGDS devices. Free ispGDS assembler software is available from the Lattice BBS at 503-693-0215 under GDSPKG.ZIP file. This software is also available on diskette by calling the Lattice Hotline at 1-800-327-8425 (FASTGAL). For design engineers who are familiar with standard third party compiler software packages, ABEL from Data I/O and CUPL from Logical Devices also support all ispGDS devices.

Using the ispGDS Compiler

The compiler will accept an ASCII text file containing the ispGDS programming instructions, and will create JEDEC and .DOC files. Once a JEDEC file has been created, the ispGDS device can be programmed by either downloading the JEDEC file to a programmer, or by using the ispGDS Download utility to program the device using the parallel port of an IBM compatible PC.

Compiler Syntax

The basic compiler syntax supports inserting comments, title, device type, pin assignments and input/output assignments. The ispGDS compiler source file comment lines are denoted with quote marks at the beginning of the comment lines. The title is defined with the key word "title =". Any text following the "title =" key word that is within single quotes is defined to be the title of the design. Similarly, the device type is defined by the key word "device =" followed by one of the three valid device types -- ispGDS22, ispGDS18, ispGDS14. The compiler syntax also allows the user to assign pin names by typing in a 10 character pin name followed by at least a single space, the "pin" key word and the pin number. This pin assignment is optional since the compiler syntax allows the user to use the "pin" key word and the pin number directly in the input/output assignments.

The output pins are assigned on the left side of the equation and the input pins are assigned on the right side of the equation. To assign an output pin to either high or low, simply assign "H" or "L" respectively on the right side of the equation. If you need to assign an input pin to multiple output pins, use one line for each assignment, as shown in the following example. In the example below,

pin 28 is an input that is routed to three outputs — pin 1, pin 2 and pin 3. Further, each output's polarity can be individually defined. The example shows pin 3 as an active low polarity whereas pin 1 and pin 2 are defined to be active high polarity. An example source file is appended at the end of this document.

```
pin 1 = pin 28
pin 2 = pin 28
!pin 3 = pin 28
```

Assembling a File

To use the assembler, create an ASCII ispGDS source file, then invoke the assembler from the DOS command line. For example:

```
gasm <test.gds>
```

where test.gds is the name of the ispGDS source file. GASM will create a JEDEC file with the same base name, and a .JED extension, like "test.jed," and a doc file with a .DOC extension, like "test.doc."

Programming the ispGDS

You can either program the ispGDS using a JEDEC file output from the ispGDS assembler, or by using the GDS_PROG routines included in the GDSPKG software package. To program the ispGDS using a programmer, follow these steps:

1. Create an ASCII ispGDS source file
2. Assemble the ispGDS file using the ispGDS assembler (GASM).
3. Download the JEDEC file created by the assembler to the programmer and program the device. The JEDEC file will have the same name as your ispGDS source file, but will have a .JED extension (for example, "test.jed").

Alternatively, you may want to program the ispGDS devices either through the parallel port of an IBM compatible PC, or through some custom hardware configuration. The routines included in the ispGDS compiler software package are configured to use the PC parallel port for programming. If you want to use a custom hardware configuration, read through the comments in GDS_PROG for information on which routines need to be modified. If you are programming using the PC, you will need an ispDOWNLOAD Cable and ISP programming interface signals on the circuit board which will plug into the printer port on your PC.

ispGDS Compiler Support

To program using the parallel port of the PC, follow these steps:

1. Create an ASCII ispGDS source file
2. Assemble the ispGDS file using the ispGDS assembler (GASM)
3. Convert the JEDEC file to ispSTREAM format by running JEDTOISP. See the documentation on JEDTOISP for further information.
4. Run ispGDS _PROG to program the device using the parallel port.

ispGDS Source Format

The following text is an example of a ispGDS source file.

```
"This is a comment (line begins with quote mark)
title = 'DIP SWITCH REPLACEMENT CONFIGURATION'

" the ispgds device type (ispgds22, ispgds18, ispgds14)
device = ispgds22

" pin names are defined as follows
pin_name pin 28

" pin 1 is an output connected to pin 28
pin 1 = pin_name
pin 2 = pin 27

" pin 3 is another output connected to pin 28
pin 3 = pin 28

" pin 5 is always high
pin 5 = h

"pin 6 is always low
pin 6 = l
pin 8 = pin 22

"! defines the inverted output for pin 9
!pin 9 = pin 20

pin 10 = pin 19
pin 12 = pin 17
pin 13 = pin 16
pin 14 = pin 15
```

Notes

If you get an error regarding "pin 0", you may have duplicated an output pin assignment (by assigning different input signals to the same output pin). Refer to the line number in the assembler error message to locate the source of the problem.



Lattice's Solution for Plug-and-Play

Introduction

As PC systems migrate from the desktop to the laptop, add-on hardware configuration becomes even more challenging. The concept of Plug-and-Play has been advanced, making hardware configuration a software controlled function. Plug-and-Play employs configuration software to automatically configure add-on hardware so it will coexist with the rest of the system. A Plug-and-Play compliant system eliminates the need for user manipulation of add-on hardware switches or jumpers prior to card installation. Lattice's in-system programmable (ISP) technology allows Plug-and-Play systems to be implemented with ease.

The traditional method to upgrade a PC with add-on cards is subject to human error and usually becomes a frustrating process of trial and error. Before installation, the PC must be powered off and partially disassembled. Then, each add-on card's unique address and interrupt levels must be set manually set by mechanical DIP switches or jumpers. Finally, the card is installed and the PC is powered up. If the first attempt is unsuccessful, the user must remove the card, reread the manual, reset the jumper or DIP switch, plug the card back in, and hope that the jumpers are set correctly. If not, the process is repeated. Mechanical components are also historically unreliable, further complicating the procedure. When the user is unsuccessful, frustration results along with lost time and effort.

ispGDS Solution

The solution to this problem is provided by Lattice's family of in-system programmable Generic Digital Switches (ispGDS), which easily replace mechanical DIP switches and jumpers. At 14, 18, and 22 outputs, even the smallest ispGDS device contains enough I/O's to replace up to two seven-bit wide DIP switches or seven jumpers. Each ispGDS output can either be set to a logic "1" or "0" to emulate a DIP switch. In addition, the ispGDS I/O's are equally divided into two banks: bank A and bank B. Any input in bank A can be connected to any or all outputs in bank B and vice-versa, effectively emulating mechanical jumpers. Jumper emulation allows devices to act as a cross switch matrix, providing 7x7, 9x9 and 11x11 matrix solutions. Furthermore, as semiconductors have no moving parts, they are more reliable than their mechanical counterparts.

System Implementation of Plug-and-Play

Although ispGDS devices can be configured either by in-system programming (ISP) or by using industry standard PLD programmers, in-system programming better satisfies the Plug-and-Play requirements of the add-on card. By programming the ispGDS devices "in-system," add-on cards can be inserted into the PC without having to manually set any DIP switches or jumpers; the configuration software for the add-on card can set the card's address and interrupt configurations. The ability to program the card using 5V TTL level signals, combined with the ispGDS C language routines from Lattice, provides the user with an easy approach to implement Plug-and-Play compliant add-on cards.

There are two system design problems to consider when programming ispGDS devices in-system. First, the ispGDS device must have its own address space so that it can be addressed and programmed anywhere on the PC bus. Unfortunately, most PC addresses are already dedicated for specific purposes per the IBM PC specification. However, some addresses are specified as read only, which means they are uncommitted for write functions. For example, the address for the game port (200-207) is specified as read only. By using an external GAL device to decode the ispGDS address to 200-207 (write only), the ispGDS can be programmed across the PC bus.

The second problem occurs when two or more PC bus add-on cards have ispGDS devices residing at the same write address. If the add-on card configuration software tries to program an ispGDS device on one of these cards, the ispGDS devices on the other cards are also addressed and programmed. This problem is remedied by using the logic in the interface GAL device and the ISP feature of the ispGDS device. Take, for example, a card with three daisy-chained ispGDS devices on it (see figure 1). By assigning three of the outputs of the first serial ispGDS device (as shown by GDSSEL0..2 in figure 1) the card's ispGDS write address can be moved within the address space. ispGDS devices come from the factory with their outputs set to the high impedance state. By using external pull-up resistors on the three ispGDS outputs dedicated for the ispGDS write address space, the three Least Significant Bit (LSB) outputs default to 111. Hence, straight from the factory, the ispGDS address space can be made to default to address 207.

Lattice Solution for Plug-and-Play

When the add-on card is inserted into the PC for the first time, the configuration software finds an unused write address, other than the default 207, and programs the ispGDS chain address to this address. This is accomplished by programming the ispGDS devices with a JEDEC fuse map, in-system, via the PC bus. This sets the dedicated ispGDS address to the desired write address value. Once the ispGDS address has been set, the address of the card, interrupt levels, and other components can be set in-system by downloading another JEDEC fuse map.

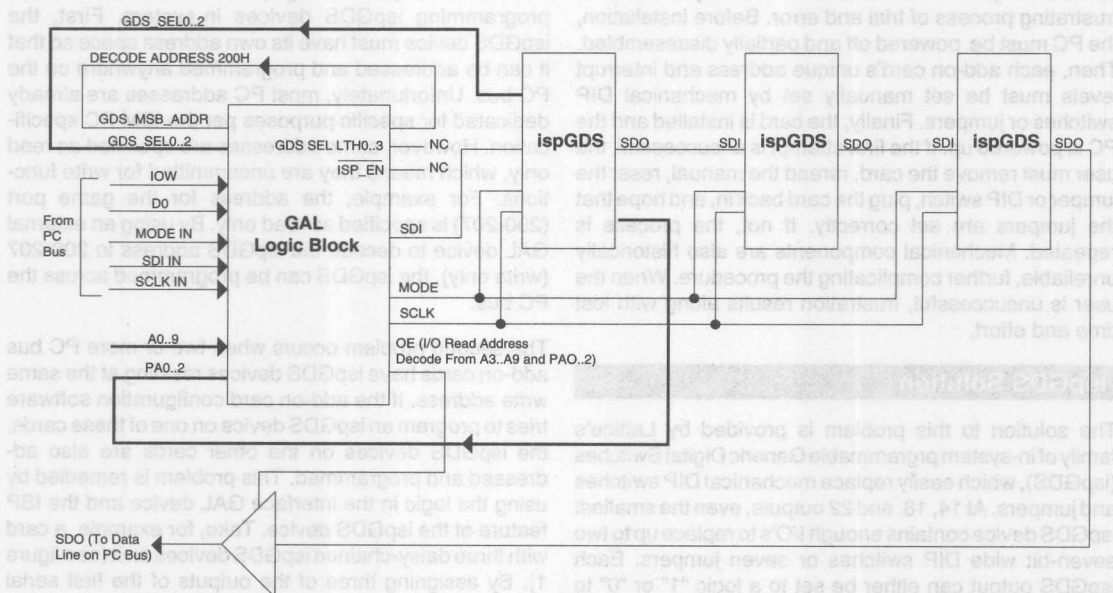
The block diagram (Figure 1) and the ABEL example file that implements the GAL decoder logic are included with this application note. The ispGDS C source code is provided by Lattice. For a detailed explanation of this

source code, please consult the ispCODE Reference Manual. Using the ABEL example file, it is easy to implement ISP on the ispGDS. Simply modify the ispGDS C source code to control data bit 0 across the PC bus; in the GAL decoder logic provided, data bit 0 acts like an ISP enable signal when written to the ispGDS write address.

Conclusion

The DIP switch and jumper approach is obsolete for Plug-and-Play add-on cards. Using minimal decoder logic, the ispGDS C source code provided by Lattice, and the in-system programming capability, ispGDS devices can effectively provide a real Plug-and-Play solution for those who exploit them.

Figure 1. ispGDS Interface Block Diagram



Decoder Logic ABEL Source File

```
module gdsdcode
title 'gds isp controller/decoder';
gdsdcode device 'p20v8' ;
iow pin 1;
a2,a1,a0 pin 2,3,4;
lsb_addr=[a2,a1,a0];
gds_msb_addr pin 23;
"a9,a8,a7,a6,a5,a4,a3 pin 5,6,7,8,9,10,11
gds_sel2,gds_sel1,gds_sel0 pin 5,6,7;
gds_sel=[gds_sel2,gds_sel1,gds_sel0];
gds_sel_ltch2,gds_sel_ltch1,gds_sel_ltch0 pin22,21,20;
gds_sel_ltch=[gds_sel_ltch2,gds_sel_ltch1,gds_sel_ltch0];
gds_addr pin 19;
isp_en pin 18 istype 'reg_d,invert';
d0 pin 8;
mode_in pin 9;
mode pin 17 istype 'reg_d,invert';
sdi_in pin 10;
sdi pin 16 istype 'reg_d,invert';
sclk_in pin 11;
sclk pin 15 istype 'reg_d,invert';
" gds_msb_addr=a9 & !a8 & !a7 & !a6 & !a6 & !a5 & !a4 & !a3
gds_lsb_addr=!((gds_sel_ltch2 $ a2) # (gds_sel_ltch1 $ a1) # (gds_sel_ltch0 $ a0));
equations
gds_sel_ltch= gds_sel & !isp_en
# gds_sel_ltch & isp_en;
gds_addr=(gds_lsb_addr & gds_msb_addr);
isp_en.d=isp_en & !gds_addr
# d0 & gds_addr;
isp_en.clk=iow;
mode.d= mode & !gds_addr
# (isp_en & mode_in) & gds_addr;
mode.clk=iow;
sclk.d= sclk & !gds_addr
# sclk_in & gds_addr;
sclk.clk=iow;
sdi.d= sdi & !gds_addr
# sdi_in & gds_addr;
sdi.clk=iow;
end;
```

Notes

Section 1: Introduction

Section 2: ispLSI and pLSI Architecture Overview

Section 3: ispLSI and pLSI Development Tools

Section 4: ispLSI and pLSI Application Notes

Section 5: GAL Architecture Overview

Section 6: GAL Development Tools

Section 7: GAL Application Notes

Section 8: In-System Programmable Generic Digital Switch (ispGDS)

Section 9: Design Techniques

User Electronic Signature	9-1
Driving CMOS Inputs with GAL Devices	9-3
Metastability Report	9-5
Latch-Up Protection	9-19

Section 10: Article Reprints

Section 11: Technology, Quality, and Reliability Overview

Section 12: General Section

Section 1: Introduction	
Section 2: iapLSI and pLSI Architecture Overview	
Section 3: iapLSI and pLSI Development Tools	
Section 4: iapLSI and pLSI Application Notes	
Section 5: GAL Architecture Overview	
Section 6: GAL Development Tools	
Section 7: GAL Application Notes	
Section 8: In-System Programmable Generic Digital Switch (ipGDS)	
Section 9: Design Techniques	
User Electronic Signature	9-1
Driving CMOS Inputs with GAL Devices	9-3
Memory Report	9-5
Latch-Up Protection	9-13
Section 10: Article Reprints	
Section 11: Technology, Quality, and Reliability Overview	
Section 12: General Section	

User Electronic Signature

Introduction

In the course of system development and production, the proliferation of PLD architectures and patterns can be significant. To further complicate the record-keeping process, design changes often occur, especially in the early stages of product development. The task of maintaining "which pattern goes into what device for which socket" becomes exceedingly difficult. What's more, once a manufacturing flow has been set, it becomes important to "label" each PLD with pertinent manufacturing information, which can be quite beneficial in the event of a customer problem or return - traceability aided by a manufacturing history can help to quickly reconstruct details of a defective product and thereby effect a speedy solution.

The Lattice GAL, ispGAL, ispGDS and ispLSI families can ease the problems associated with document control and traceability, thanks to a feature called User Electronic Signature (UES). This brief describes the concept behind the UES, how it is used, and the advantages associated with manufacturing flow control, documentation, and traceability.

The UES is basically a user's "notepad" provided in electrically erasable (E²) cells on each GAL, ispGAL, ispGDS and ispLSI device. Essentially an extra row that's appended to the array and allocated for data storage, the physical size of the UES varies by device type. The table below indicates the various sizes of the UES.

Device	UES Size
GAL16V8/20V8	64 bits
GAL16VP8/20VP8	64 bits
GAL16V8Z/20V8Z	64 bits
GAL16V8ZD/20V8ZD	64 bits
GAL18V10	64 bits
GAL22V10	64 bits
GAL26CV12	64 bits
GAL20XV10	40 bits
GAL20RA10	64 bits
GAL6001/6002	72 bits
ispGAL22V10	64 bits
ispGDS	32 bits
ispLSI 1016	64 bits
ispLSI 1024	104 bits
ispLSI 1032	144 bits
ispLSI 1048	224 bits

Lattice incorporated the UES to store such design and manufacturing data as the manufacturer's ID, programming date, programmer make, pattern code, checksum, PCB location, revision number, and product flow. The intent was to assist users with the complex chore of record maintenance and product flow control. In practice, the UES can be used for any of a number of ID functions.

Within the various number of bits available for UES data storage, users may find it helpful to define specific fields to make better use of information storage. A field may use only one bit (or all bits), and may contain a variety of topics. Some fields should probably be reserved for future expansion. The possibilities for fields are endless, and completely up to the user. As an example for the GAL16V8, the UES could be divided into five fields: manufacturer's ID (2 Bytes or 16 Bits), device program data code (2 Bytes), programmer ID code (1 Byte), pattern ID code (2 Bytes), and a reserved section (1 Byte).

Even with the device's security feature enabled, the UES can still be read. If a pattern code were stored in the UES, the user could always identify which pattern had been used in a given device. In this way, a device pattern could be confidentially retrieved. As a second safety feature, when a device is erased and repatterned, the UES row is automatically erased. This prevents any situation in which an old UES might be associated with a new pattern (no information is better than wrong information). It is the user's responsibility to update the UES when reprogramming. It should be noted that UES information will be included in the checksum reading. Therefore when the UES is modified the checksum will also change.

The UES may be accessed (read or write) through one of three methods. First, most third party programmers support the UES option through the programmer's user interface, so programming or verifying the UES is as simple as programming or verifying any other array. Second, the UES may be installed within the JEDEC file by selecting the proper fuse locations in the fuse map. Please consult the latest Lattice Data Book for the fuse locations of the UES. Third, the UES can be written or read using Lattice's ispCODE software with routines provided in the ispCODE library. Further information on using ispCODE software to program the UES can be found in the latest Lattice Data Book.

Though provided to assist the designers and manufacturers who utilize Lattice products, making use of the UES is not essential to enjoying the many benefits of our

User Electronic Signature

devices. For those willing to invest in it, however, the reduction of "hidden costs" associated with PLDs can be significant. The following outlines some of the opportunities presented; the reader is referred to the brief "Hidden Costs in PLD Usage" in this handbook to ascribe the value of each benefit.

Eliminating Labels

By automatically storing the appropriate identification information into device UES locations while the programming hardware is patterning the device, the need for a costly additional handling step to apply messy gummed labels or ink is eliminated. What's more, throughput and quality of the patterned devices are greatly increased.

Document Control

The job of document control becomes more manageable when using the device UES, since a pattern code in the UES can specify each pattern and its application. This proves an absolute boon in Military programs, where accurate documentation is essential. If a change occurs, it is easily handled with a new pattern code. In fact, with a pattern code in each device, a readout can actually be conducted during board assembly. Code verification would ensure the use of properly patterned devices and serve as a quality-monitor step. Moreover, validation is simplified when checking against a lot or board-traveler, since master devices are not required.

Software Revisions

With the UES, a software ID code can be stored and referenced in Document Control to a current pattern version. When a revision occurs, a new pattern code is simultaneously stored in the UES. Pattern codes can be monitored to verify that incorrect versions of software are not inadvertently being used. Since Lattice devices are reprogrammable, any material flagged with an improper pattern code can simply be sent back and reprogrammed to the current pattern revision. Also, when security is enabled, a UES-resident pattern ID code is the only certain means of documenting which pattern resides within a device.

Manufacturing Information

As described earlier, manufacturing information stored in the UES can help track down problems, should product be serviced in the field or returned. A field technician could easily read checksum and pattern revision information to facilitate rapid debug assuming these fields were stored in the UES. Additionally, if each board-assembly location were coded into the devices used at that assembly site, customer board returns might be linked to a common source.

Manufacturing Flow

With the UES, devices can all be preprogrammed at one location and given a destination code. Upon shipment and receipt, sample readouts of destination codes could be performed to ensure that the proper devices were received.

As systems become more complex, production and document control costs can become dominant. UES is one of the many valuable ease-of-use features offered in the Lattice GAL, ispGAL, ispGDS and ispLSI families that can tame such costs.

Device	UES Size
GAL16V820VB	64 bits
GAL16P820VP8	64 bits
GAL16V820VBZ	64 bits
GAL16V820VBZD	64 bits
GAL16V10	64 bits
GAL22V10	64 bits
GAL28CV12	64 bits
GAL20XV10	40 bits
GAL20P10	64 bits
GAL8001600S	72 bits
ispGAL22V10	64 bits
ispGDS	32 bits
ispLSI 1018	64 bits
ispLSI 1024	104 bits
ispLSI 1032	144 bits
ispLSI 1048	224 bits

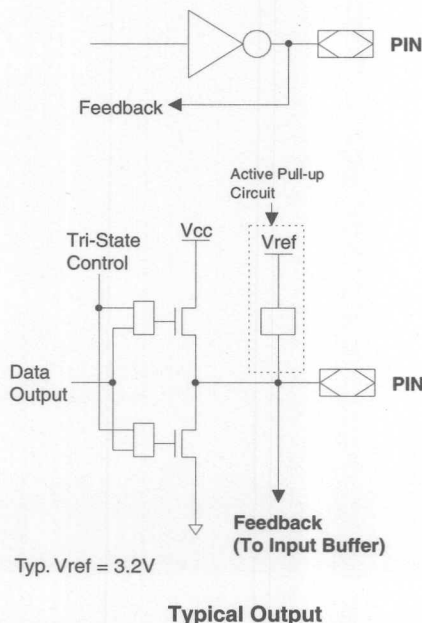
Driving CMOS Inputs with GAL Devices

Introduction

While Lattice GAL[®] devices do not have a true CMOS output structure, in most cases they are able to reliably drive CMOS inputs. GAL devices are designed with TTL-level input and output specifications. There are two reasons for this. First, because Lattice GAL devices are as fast, or faster than the fastest equivalent bipolar devices, they are often used as bipolar replacements. While a design may initially be implemented with bipolar devices, often the designer has the opportunity to replace the bipolar device with the lower-power and better tested GAL device. In these cases, the GAL device must drop into the same socket with identical functionality. Second, switching noise is greatly reduced by using TTL-level outputs. Switching an output from Vcc to Ground will generate considerably more noise than switching from a TTL high to a TTL low.

NMOS Outputs

GAL devices use a NMOS output structure, which does not allow the output signal to go to the rail but still gives plenty of margin to TTL specs. The NMOS output structure also completely eliminates any possibility of latch-up.



Under typical conditions of room temperature and nominal Vcc, GAL devices will exhibit a V_{OH} of about 4.2 volts. This value will change somewhat with temperature, Vcc, and normal process variations. Process and temperature are the most important factors, in that they affect the amount of voltage drop between Vcc and the output pin. Therefore the most valuable way to specify a V_{OH} value is to specify the difference between Vcc and V_{OH}. In this manner, a designer with greater control over Vcc can know exactly what the true worst-case V_{OH} value will be. The following tables show the V_{OH} values that can be expected under different conditions.

One factor that helps to make it all work is that even though the output voltage or the GAL device will drop with Vcc, the input transition point of the CMOS devices being driven will also drop.

Using pull-up resistors on the outputs of the GAL device will also help to assure proper CMOS output levels. A 10 Kohm pull-up resistor will pull a GAL device's output to the rail. Of course the time required to do so depends on the total capacitance on the output pin, which includes the I/O capacitance of the GAL device output, the input capacitance of the devices being driven, and the parasitic capacitances on the board.

As for the GAL16/20V8Z and GAL16/20V8ZD zero-power devices, the DC specification guarantee the CMOS output specification at I_{OH} of -100μA at V_{OH} of Vcc-1V. These devices will be able to drive CMOS inputs without the pull-up resistors on the output of the GAL devices.

Commercial and Industrial Devices

Specification	Condition	Min. Value
V _{OH}	I _{OH} = -3.2 mA	2.4 V

Military Devices

Specification	Condition	Min. Value
V _{OH}	I _{OH} = -2.0 mA	2.4 V

Under typical conditions of room temperature and nominal V_{CC}, GAL devices will exhibit a V_{OH} of about 4.5 volts. This value will change somewhat with temperature, V_{CC}, and normal process variations. Process and temperature are the most important factors in that they affect the amount of voltage drop between V_{CC} and the output pin. Therefore the most valuable way to specify a V_{OH} value is to specify the difference between V_{CC} and V_{OH}. In this manner, a designer with greater control over V_{CC} can know exactly what the true worst-case V_{OH} value will be. The following table shows the V_{OH} values that can be expected under different conditions.

One factor that helps to make it all work is that even though the output voltage of the GAL device will drop with V_{CC}, the input transition point of the CMOS devices being driven will also drop.

Using pull-up resistors on the outputs of the GAL device will also help to assure proper CMOS output levels. A 10 kΩ pull-up resistor will pull a GAL device's output to the rail. Of course the time required to do so depends on the total capacitance on the output pin, which includes the VO capacitance of the GAL device output, the input capacitance of the devices being driven, and the parasitic capacitances on the board.

As for the GAL16V08B2 and GAL16V08B2 zero-power devices, the DC specification guarantees the CMOS output specification at I_{OH} of -100 μA at V_{OH} of V_{CC} - 1 V. These devices will be able to drive CMOS inputs without the pull-up resistors on the output of the GAL device.

Commercial and Industrial Devices

Specification	Condition	Min. Value
V _{OH}	I _{OH} = -3.5 mA	2.4 V

Military Devices

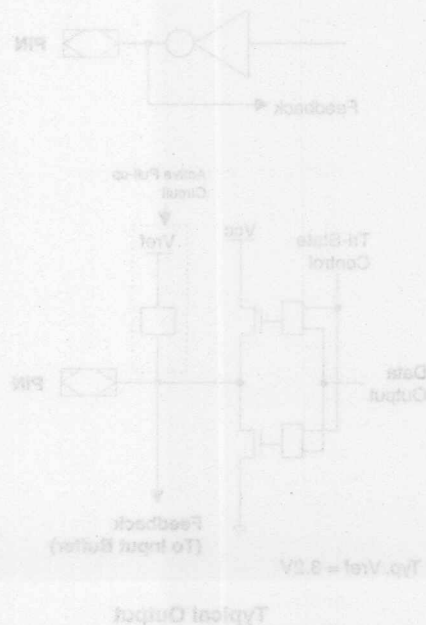
Specification	Condition	Min. Value
V _{OH}	I _{OH} = -2.0 mA	2.4 V

Introduction

While Lattice GAL[®] devices do not have a true CMOS output structure, in most cases they are able to reliably drive CMOS inputs. GAL devices are designed with TTL-level input and output specifications. There are two reasons for this. First, because Lattice GAL devices are as fast, or faster, than the fastest equivalent bipolar devices, they are often used as bipolar replacements. While a design may initially be implemented with bipolar devices, often the designer has the opportunity to replace the bipolar device with the lower-power and better tested GAL device. In these cases, the GAL device must drop into the same socket with identical functionality. Second, switching noise is greatly reduced by using TTL-level outputs. Switching an output from V_{CC} to Ground will generate considerably more noise than switching from a TTL high to a TTL low.

CMOS Outputs

GAL devices use a NMOS output structure, which does not allow the output signal to go to the rail but still gives plenty of margin to TTL space. The NMOS output structure also completely eliminates any possibility of latch-up.



Metastability Report

Introduction

The dictionary definition of metastability is "a situation that is characterized by a slight margin of stability." When applied to bi-stable (digital) logic, the term refers to an undesirable, marginally stable output state between V_{IL} max and V_{IH} min.

Metastability can occur in bi-stable storage elements (registers, latches, memories, etc.) when setup and/or hold times are violated. Since setup and hold times vary with temperature and operating voltage, among other factors, the times referred to here are not the min/max numbers printed in data sheets, but rather the actual times for the given set of operating conditions. Typical applications where such times are likely to be violated include bus and memory arbiters, interfaces, synchronizers, and other state machines employing asynchronous inputs or asynchronous clocks.

Metastability manifests itself in a number of different ways. Common responses are (shown as they might be captured on a digital oscilloscope in Figure 1): runt pulse (1a), decreased output slew rate (1b), output oscillation (1c), and increased clock-to-output time (1d). By definition, the phenomenon of metastability is statistical in nature. Not only is entry into the metastable state uncertain, but the time spent there can also vary.

Because PLDs are commonplace in today's designs, a thorough understanding of their metastable behavior is crucial. In some applications, output anomalies shorter than one clock cycle may be acceptable, but in applications where the register output is used as a control signal (clock, bus grant, chip select, etc.) for other circuitry, faults such as runt pulses and oscillation cannot be tolerated.

This report will not study the causes or characteristics of metastability in great detail; excellent material has already been prepared on this subject [1-5]. Rather, this report will introduce a mathematical model for the metastable phenomenon, discuss potential test methodologies, present and compare test results from various bipolar and CMOS PLDs, and discuss how to interpret the data. This report will close with suggestions on how to design metastable tolerant systems.

Derivation of Constants

The basic premise of all metastability models is that a device's output is more likely to have settled to a valid

state in time(t) than in time(t-n). In fact, the failure probability distribution follows an exponential curve. Figure 2 shows a typical failure frequency plot.

It is accepted [1] that metastable failures can be accurately modeled by the equation:

$$\log \text{Failure} = \log \text{MAX} - b(\Delta - \Delta_0) \quad (1)$$

In this equation, MAX represents the maximum failure rate for a particular environment, Δ is the time delayed before sampling the DUT (Device Under Test) output, and Δ_0 is the time at which the number of failures starts to decrease. On a failure frequency plot (such as the one in Figure 2), Δ_0 represents the knee of the curve. The constant b is the rate at which the frequency of failures decreases after the knee is reached.

Recall that:

$$\log X = a \ln(X), \text{ where } a = \log(e)$$

Substituting this into (1):

$$a \cdot \ln \text{Failure} = a \cdot \ln \text{MAX} - b(\Delta - \Delta_0) \quad (2)$$

MAX is related to the clock frequency (fCLOCK) and data frequency (fDATA). That is,

$$\text{MAX} = (k1 \cdot \text{fCLOCK} \cdot \text{fDATA}) \quad (3)$$

Substituting (3) into (2) and applying some algebra:

$$a \cdot \ln \text{Failure} = a \cdot \ln (k1 \cdot \text{fCLOCK} \cdot \text{fDATA}) - b(\Delta - \Delta_0)$$

$$\ln \text{Failure} - \ln (k1 \cdot \text{fCLOCK} \cdot \text{fDATA}) = -b/a(\Delta - \Delta_0)$$

Setting $k2 = b/a$ and rearranging the equation yields:

$$\text{Failure} = (k1 \cdot \text{fCLOCK} \cdot \text{fDATA})e^{-k2(\Delta - \Delta_0)} \quad (4)$$

When used with equation (4), the constants $k1$, $k2$, and Δ_0 , completely describe a particular device's metastable characteristics; they indicate how quickly a device can resolve the metastable condition. Devices which transition out of the metastable region quickly are characterized by a small Δ_0 and a large $k2$.

The constant $k1$ is peculiar to the test apparatus (it can be thought of as a "scaling factor"). The maximum metastable failure rate (MAX) is limited by fCLOCK; a failure cannot occur if the device isn't clocked. Likewise, it is true that a metastable failure cannot occur unless data has changed. So, if $\text{fDATA} < \text{fCLOCK}$, then MAX

Metastability Report

= fDATA. This was the case in the test fixture Lattice used (fCLOCK=10MHz, fDATA=2.5MHz). Substituting MAX = fDATA back into equation (3) yields: $k1 = 1 / fCLOCK$, so $k1 = 100ns$ for our tests.

Test Fixture

The goal of testing a particular device's metastable characteristics is to generate real numbers for the constants $k2$ and Δo . To do this, the device must first be forced into the metastable state. This is done by intentionally violating setup and/or hold times. Once metastable, the output can be observed on an oscilloscope or used to increment an event counter.

Traditional Approach

One approach to characterizing a device's metastable behavior employs a test fixture similar to that shown in Figure 3a. In such a fixture, data to the device includes a "jitter band" so that the device sees changing data as it is clocked. The DUT output is fed to a window comparator to determine when it is in the metastable region (between V_{IL} max and V_{IH} min). The comparator output can be sampled periodically and used to increment an event counter.

This method of testing, though it directly yields MTBF numbers, has some drawbacks. The first is that it does not distinguish between the different types of metastable behavior (runt pulse, oscillation, slow rise/fall time, delayed transition), and it may have difficulty

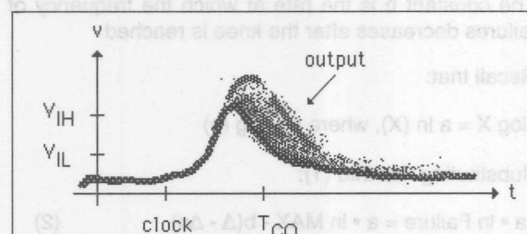


Figure 1a. Runt Pulse

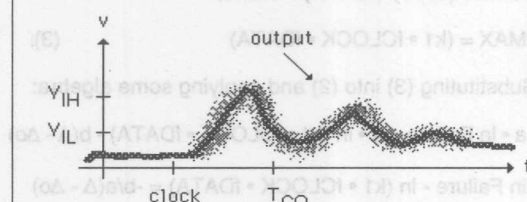


Figure 1c. Output Oscillation

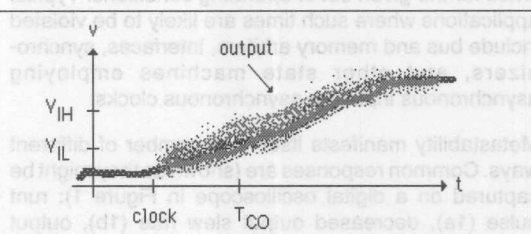


Figure 1b. Decreased Slew Rate

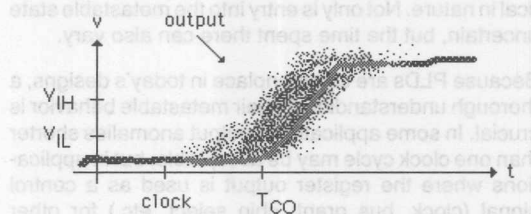


Figure 1d. Increased T_{CO}

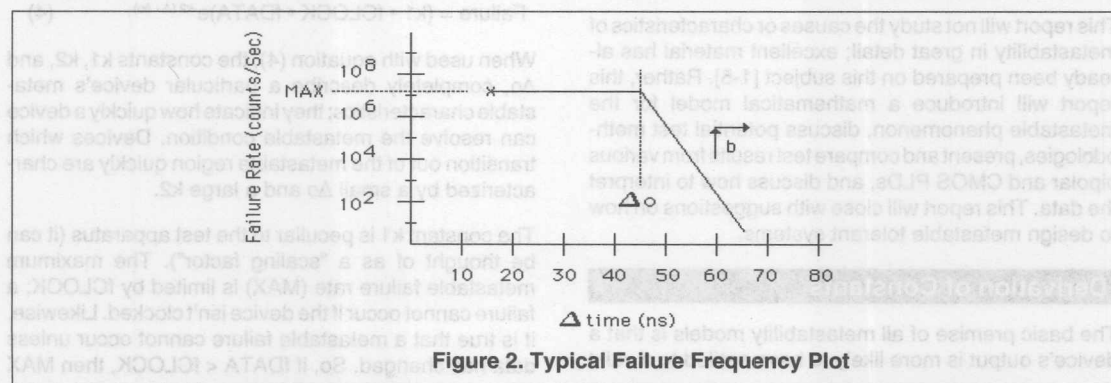


Figure 2. Typical Failure Frequency Plot

Metastability Report

detecting every type. Also, the registers used in the detector circuit itself may become metastable, which would adversely affect the results.

A New Approach

The test method used to gather data for this report used the circuit shown in Figure 3b. The tester employed an "infinite precision" variable delay circuit to control clock placement with respect to data. This arrangement allowed exact worst case placement of the clock, so as to induce metastability with nearly every clock pulse.

Using a digital oscilloscope (Tektronix 11403A) in point accumulate mode, metastable failures were recorded over a lengthy period of time. A hardcopy was then made and the constants empirically obtained (details below).

The oscilloscope approach, being visual in nature, enables the designer to make educated decisions re-

garding maximum clock and data rates, as well as the suitability of using the output to drive other circuitry. The five minute sample period used in our tests contained approximately 750 million failures. Much longer sample periods were evaluated, but they provided no perceptible gain in usable information.

A slight disadvantage of this approach is that extracting k_2 and Δ_0 values from the hardcopies is not straightforward. Because each point on the hardcopy can represent any number of actual samples (between one and 1.5 million), one cannot simply count the points at time(t) for the MTBF at that time (although, in the case of the scattered points, the probability is low that a single isolated point represents more than one sample).

To generate values for k_2 and Δ_0 , it was necessary to refer to previous metastability studies [1]. By studying the output plots of devices with known constants, certain relationships were established. For example, it was determined that Δ_0 represents the time from the leading

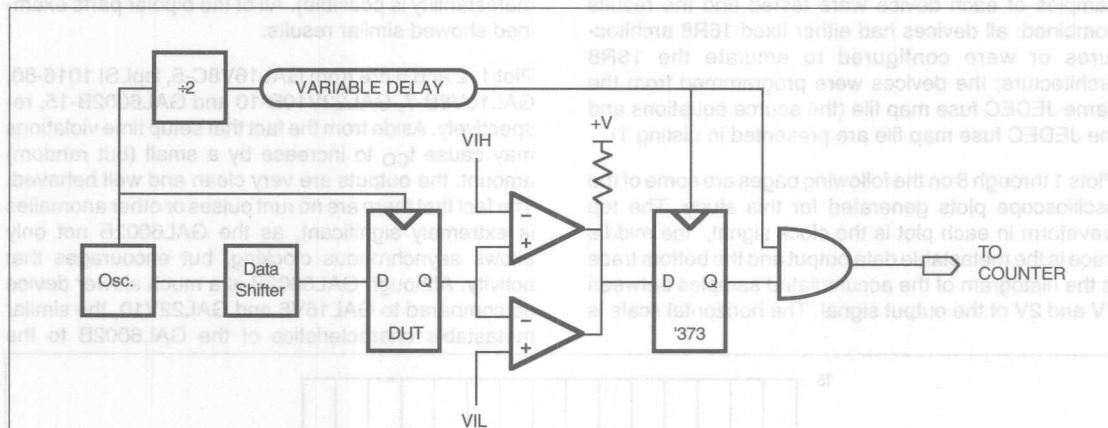


Figure 3a. Traditional Metastability Test Circuit

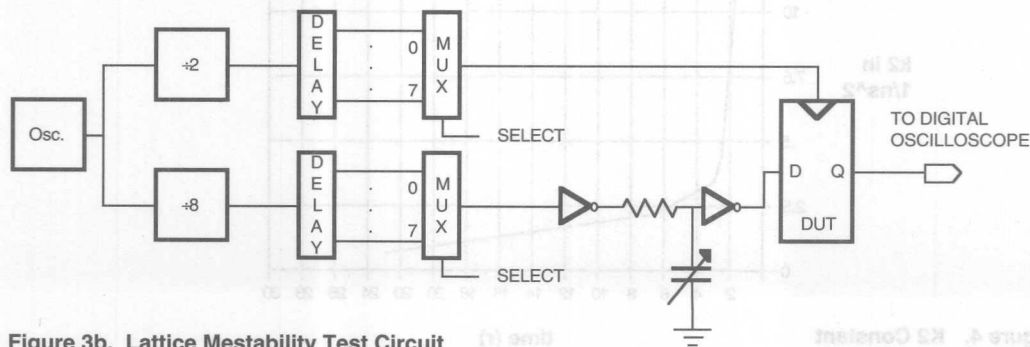


Figure 3b. Lattice Mestability Test Circuit

Metastability Report

edge of the output until the "dot density" starts to decrease measurably. It should be noted that Δo in previous studies included device propagation delays, whereas in our test it does not.

The time from Δo until the dot density equals zero was defined to be the "time to metastable release" or simply time(r). The relationship between k2 and time(r) is given below in (5), and shown graphically in Figure 4. Recall that $MAX=2.5 \times 10^6$ and $a=\log(e)$.

$$k2 = \log(MAX) / (\text{time}(r) \cdot a) = 14.73/\text{time}(r) \quad (5)$$

Interpreting the Results

In addition to examining E²CMOS GAL devices, this study also tested several bipolar PAL devices as well as other CMOS PLDs. To insure that the results of this study would be relevant, all necessary precautions were observed: the devices were of recent vintage and were acquired blindly through distributors; multiple samples of each device were tested and the results combined; all devices had either fixed 16R8 architectures or were configured to emulate the 16R8 architecture; the devices were programmed from the same JEDEC fuse map file (the source equations and the JEDEC fuse map file are presented in Listing 1).

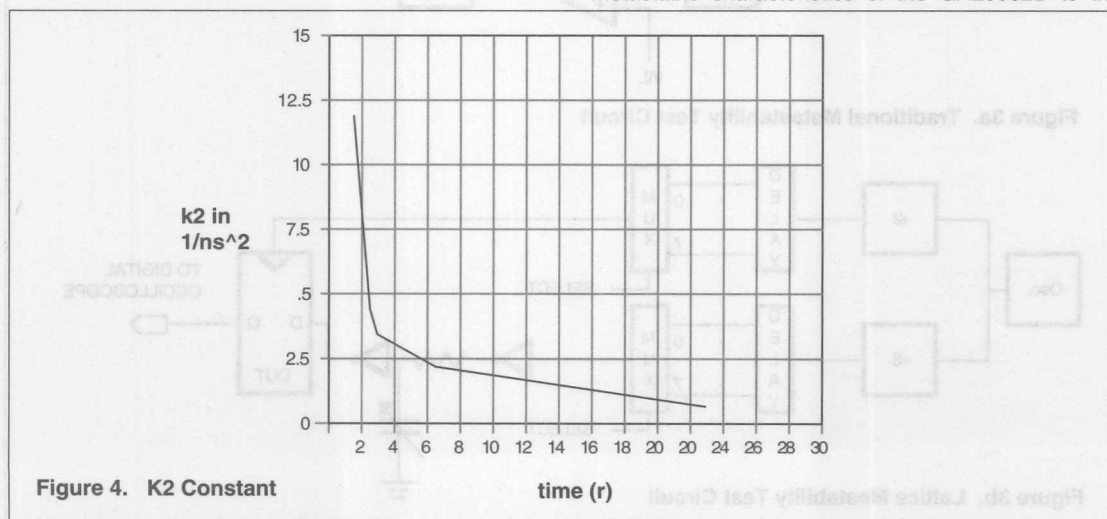
Plots 1 through 8 on the following pages are some of the oscilloscope plots generated for this study. The top waveform in each plot is the clock signal, the middle trace is the metastable data output and the bottom trace is the histogram of the accumulated samples between 1V and 2V of the output signal. The horizontal scale is

2ns per division, so the exact clock to output time of the metastable output condition can be read directly. The vertical scale is 2V per division for the top trace, and 1V per division for the middle trace.

The middle waveform in each plot is the metastable device output which is the only signal captured in point accumulate mode. In every case, the output signal plot shows two stable levels after the transition. This is a direct result of the "indecision" caused by metastability; on some cycles the output settled to a high level, while on others it settled to a low level.

Plot 4 shows the response of a bipolar PAL16R8-7. Notice the very well defined runt pulse (this correlates with previous data gathered on similar devices by the manufacturer [1]). The absence of a secondary trace along ground indicates that the output always starts to transition to a high level, even when it finally settles to a low level. This characteristic makes the device unsuitable for use in control path applications (when metastability is possible). All of the bipolar parts examined showed similar results.

Plot 1, 2 and 5 are from GAL16V8C-5, ispLSI 1016-80, GAL16V8B-7, GAL22V10B-10 and GAL6002B-15, respectively. Aside from the fact that setup time violations may cause t_{CO} to increase by a small (but random) amount, the outputs are very clean and well behaved. The fact that there are no runt pulses or other anomalies is extremely significant, as the GAL6002B not only allows asynchronous clocking, but encourages that activity. Although GAL6002B is a much slower device as compared to GAL16V8 and GAL22V10, the similar metastable characteristics of the GAL6002B to the



much faster GAL devices indicate that the inherent metastable characteristics of all the GAL devices have consistently desirable characteristics across all speed grades. Comparing Plot 1 through 5 with Plot 6 and 7 shows that characteristics of the GAL devices are superior to those of bipolar PLDs. Plot 8 illustrates metastable characteristics of the TTL flip-flop (TISN74AS74).

For reference purposes, Plots 9 through 11 are included. Plot 9 shows a normal (ie. non-metastable) GAL16V8B-7 transition, and Plot 10 a normal PAL16R8-7 transition. Plot 11 is the normal transition of the TTL flip-flop (TI SN74AS74). For consistency, only rising edges have been shown. Our tests also covered falling edges which, in general, were interesting but did not provide any additional information.

For a more quantitative look at the phenomenon of metastability, refer to the table beneath each plot. These tables list the measured values of the constants Δ_0 and k_2 for the device whose plot is shown, and for similar devices. Recall that large k_2 and small Δ_0 values are desirable. The numbers in the tables correlate closely with the results of earlier tests [1,5], confirming the validity of our test method.

Since all current GAL devices possess very similar register and output buffer circuitry, and all are fabricated using the same basic process, the data shown in Table 1 for the GAL16V8 is considered applicable to all devices and speed grades in the GAL family.

Using the Results

If a register enters the metastable state in a system, then data was obviously unstable as the register was being clocked. The argument over which data should have been captured (old or new) is academic as the register will randomly pick one or the other. Signals in most asynchronous systems are active for more than one clock cycle, so if they are missed initially, they could be captured on a subsequent clock cycle.

It is the task of the state machine designer to take adequate precautions against metastability causing illegal states to be entered. One way to do this is by using "gray codes" when ordering states. Gray code state equations allow only one state bit to change during a state transition. Thus, the worst metastability could do would be to delay a state transition by one clock cycle. If more than one bit were allowed to change, the outcome would be purely random, and probably illegal. Figure 5 shows examples of both cases.

Other solutions are to externally (or internally) synchronize the asynchronous signals, or to increase cycle times to allow time for metastable outputs to settle. An example of the latter solution is given below.

It is worth noting at this point that state machines (synchronous or asynchronous) can fail for reasons other than metastability. A not insignificant component of a PLD's specified setup time is directly attributable to internal data skewing [2]. Data skewing is the inevitable result of differing signal path lengths, loading conditions, and gate delays. Stated another way, each input to output path has its own set of actual AC specifications. If insufficient setup time has passed, different "versions" of the same data may be present at the inputs of different registers as they are clocked. A good example of this is:

```
Output_Pin19 := Input_Pin2;  
Output_Pin15 := !Input_Pin2;
```

If clocked at precisely the right moment after an input transition, one register will capture old data while the other captures new data, resulting in a system failure. This condition, though also the result of a setup time violation, should not be confused with metastability (the "incorrect" data that is captured has normal output characteristics); it is, pure and simply, the result of a violation of specifications.

Example

To determine the maximum clock rate (given an acceptable error rate) that a particular device will allow in an asynchronous environment, equation (4) is used. For example, the system shown in Figure 6 utilizes a 9600 baud (bits/sec) asynchronous data stream. The system clock period is $t_{CO} + t_{PD} + t_{SU} + \Delta$. For one failure per year:

$$3.2 \times 10^{-8} = [(1 \times 10^{-7}) / (\Delta + 22)] (9600) e^{-4(\Delta / .44)}$$

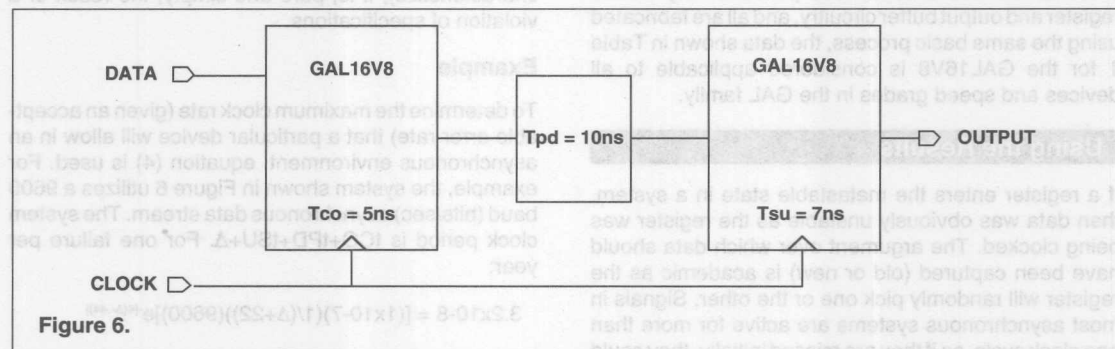
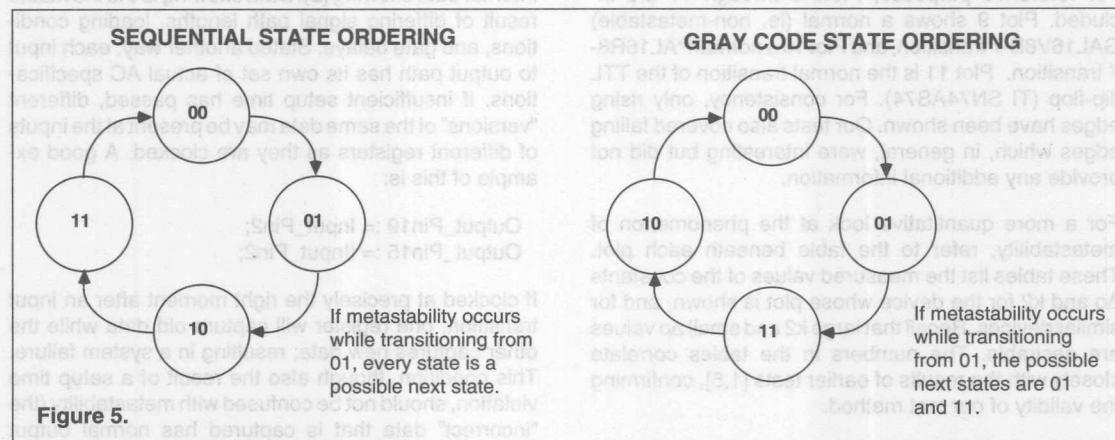
Solving for Δ yields $\Delta = 2.22$ ns, or about 2 ns, for a cycle time of 24 ns. Referring back to Plot 1, the additional delay of 2 ns intuitively makes sense. Remember, in terms of setup and hold time violations, the oscilloscope plots were made under worst case failure conditions; the scattered dots could represent MTBFs of days, years, or even millenniums in a typical asynchronous environment.

Due to the extremely quick metastable settling times of GAL devices, a relatively small increase in the cycle time will produce a dramatic improvement in reliability.

Metastability Report

Bibliography

1. D.M.Tavana (MMI), "Metastability - A study of the Anomalous Behavior of Synchronizer Circuits," in: Programmable Array Logic Handbook, Monolithic Memories Inc., 1986, pp 11-13 - 11-16.
2. K.Rubin (Force Computers), "Metastability Testing in PALs," Wescon/87 Conference Record (San Francisco, November 17-19, 1987). Los Angeles: Electronics Conventions Management, Inc, 1987, pp 16/1 1-10.
3. K.Nootbaar (Applied Microcircuits Corp.), "Design, Testing, and Application of a Metastable Hardened Flip-Flop," *ibid.*, pp 16/2 1-9.
4. J.Birkner (MMI), "Understanding Metastability," *ibid.*, pp 16/3 1-3.
5. R.K.Breuninger, K.Frank, "Metastable Characteristics of Texas Instruments Advanced Bipolar Logic Families," application note SDAA004, Texas Instruments, 1985.



<pre> MODULE metastable TITLE 'Metastable Test Pattern' u00 Device 'P16R8'; d PIN 2; q1,q2 PIN 12,19; EQUATIONS q1 := d; q2 := d; End metastable </pre> <p>Listing 1a. Source equations</p>	<p>JEDEC file for: P16R8</p> <p>Metastability Test Pattern*</p> <p>QP20* QF2048* F0*</p> <p>L0000 10111111111111111111111111111111*</p> <p>L1792 10111111111111111111111111111111*</p> <p>C07F4*</p> <p>Listing 1b. JEDEC file</p>
---	---

Metastability Report

Listing 2. ispLSI 1016 Metastability Test Source Equation from Lattice pDS Software

```
//  
// metastbl.ldf generated using Lattice pDS Version 2.20
```

```
LDF 1.00.00 DESIGNLDF;  
DESIGN metastbl;  
REVISION 00;  
AUTHOR ;  
PROJECTNAME METASTABILITY STUDY;  
PART pLSI1016-80LJ;  
OPTION Y1_AS_RESET ON;  
DECLARE  
END; //DECLARE  
SYM GLB B7 1 ;  
SIGTYPE IMOUT REG OUT;  
SIGTYPE IROUT REG OUT;  
EQUATIONS
```

```
IMOUT.CLK=ICLK;  
IROUT.CLK=ICLK;  
IMOUT.D = IMIN;  
IROUT.D = IRIN;
```

```
END;  
END;
```

```
SYM IOC IO31 1 ;  
XPIN IO MOUT LOCK 10;  
OB11 (MOUT,IMOUT);  
END;
```

```
SYM IOC IO30 1 ;  
XPIN IO ROUT LOCK 9;  
OB11 (ROUT,IROUT);  
END;
```

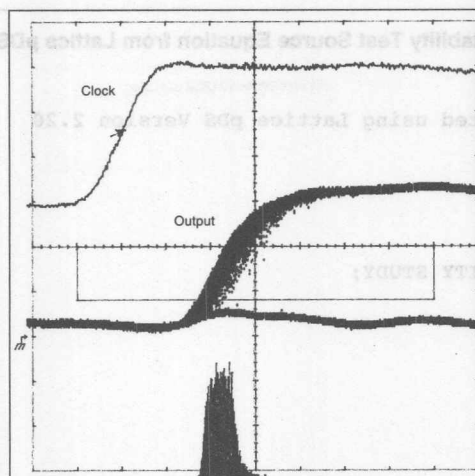
```
SYM IOC IO29 1 ;  
XPIN IO MIN LOCK 8;  
IB11 (IMIN,MIN);  
END;
```

```
SYM IOC IO28 1 ;  
XPIN IO RIN LOCK 7;  
ID11 (IRIN,RIN,ICLK);  
END;
```

```
SYM IOC Y2 1 ;  
XPIN CLK XCLK LOCK 33;  
IB11 (ICLK,XCLK);  
END;  
END; //LDF DESIGNLDF
```


Metastability Report

1V/div

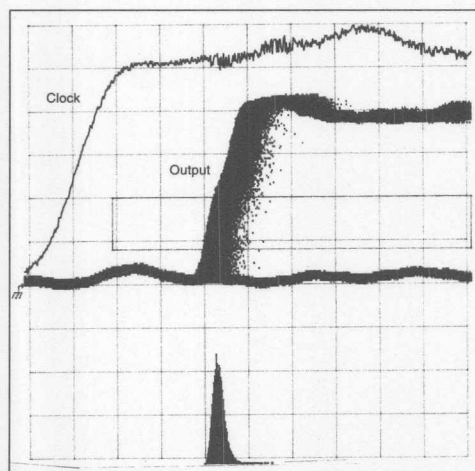


2ns/div

Plot 1. GAL16V8C-5 Metastable Output

Part #	Manufacturer	Δo (ns)	$k2$ (1/ns ²)
GAL16V8C-5	Lattice	1.4	9.82

1V/div

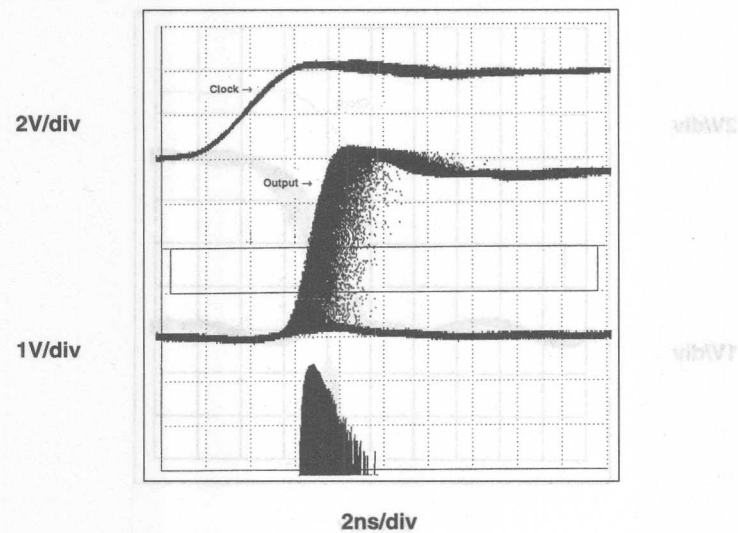


2ns/div

Plot 2. ispLSI 1016-80 Metastable Output

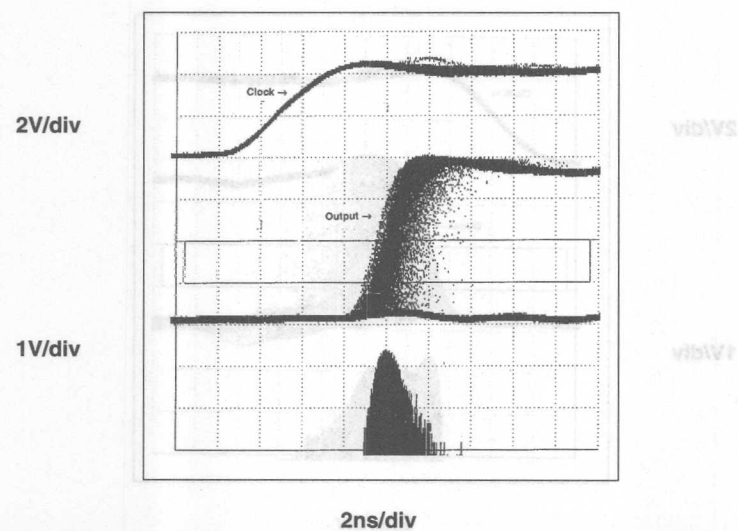
Part #	Manufacturer	Δo (ns)	$k2$ (1/ns ²)
ispLSI 1016-80	Lattice	.854	11.0

Metastability Report



Plot 3. GAL16V8B-7 Metastable Output

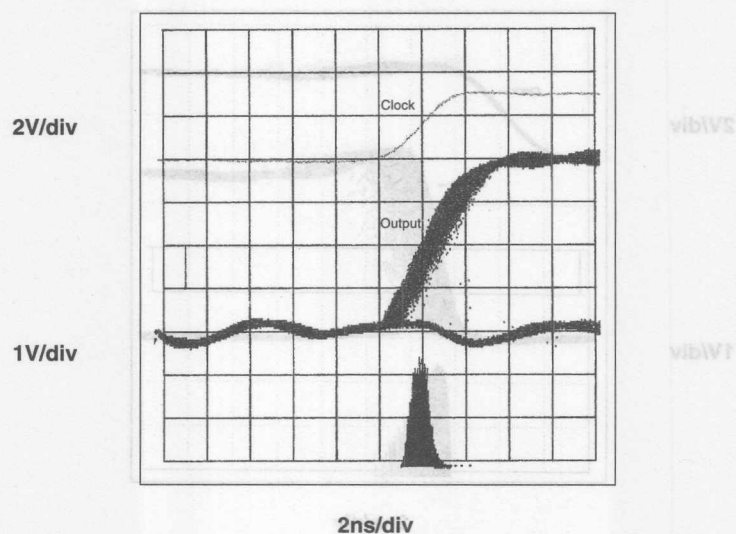
Part #	Manufacturer	Δo (ns)	k2 (1/ns ²)
GAL16V8B-7	Lattice	.44	5.0



Plot 4. GAL22V10B-10 Metastable Output

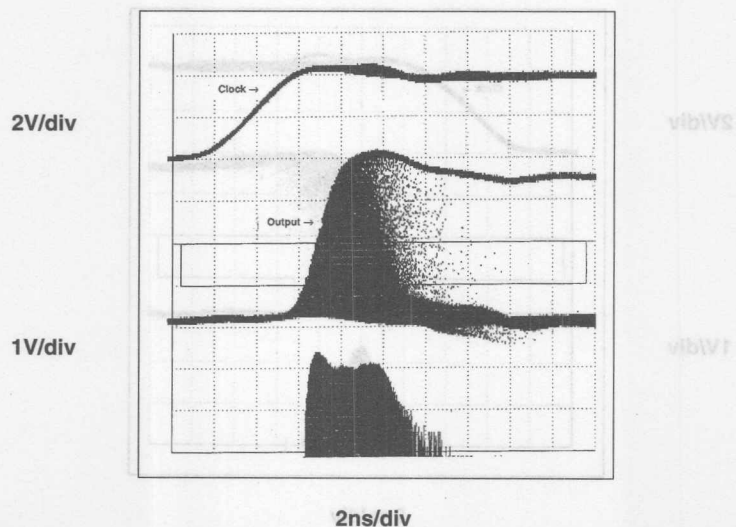
Part #	Manufacturer	Δo (ns)	k2 (1/ns ²)
GAL22V10B-10	Lattice	.51	5.2

Metastability Report



Plot 5. GAL6002B-15 Metastable Output

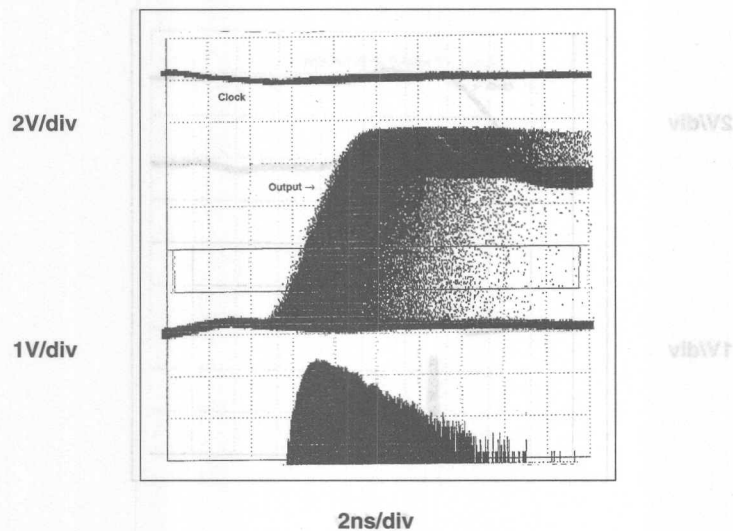
Part #	Manufacturer	Δo (ns)	k2 (1/ns ²)
GAL6002B-15	Lattice	1.1	6.52



Plot 6. PAL16R8-7 Metastable Output

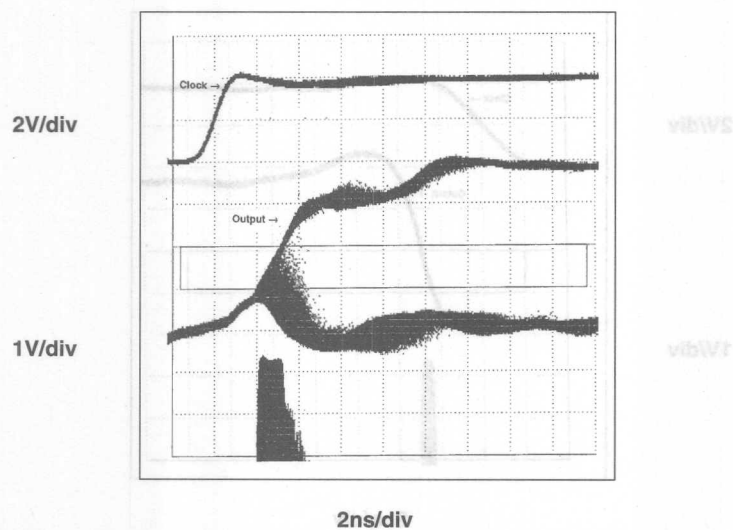
Part #	Manufacturer	Δo (ns)	k2 (1/ns ²)
PAL16R8-7	AMD	1.2	2.5

Metastability Report



Plot 7. TIBPAL16R6-7 Metastable Output

Part #	Manufacturer	Δo (ns)	k2 (1/ns ²)
TIBPAL16R6-7	TI	1.5	1.5

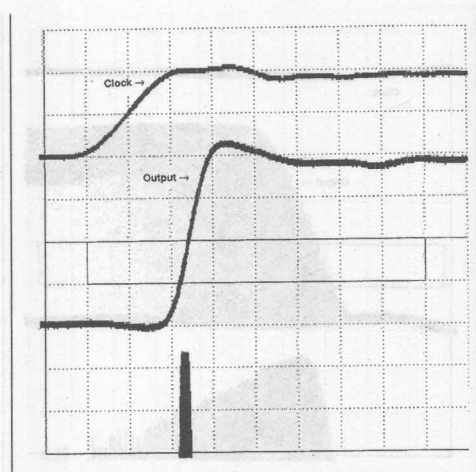


Plot 8. SN74AS74 Metastable Output

Part #	Manufacturer	Δo (ns)	k2 (1/ns ²)
SN74AS74	TI	.91	3.5

2V/div

1V/div



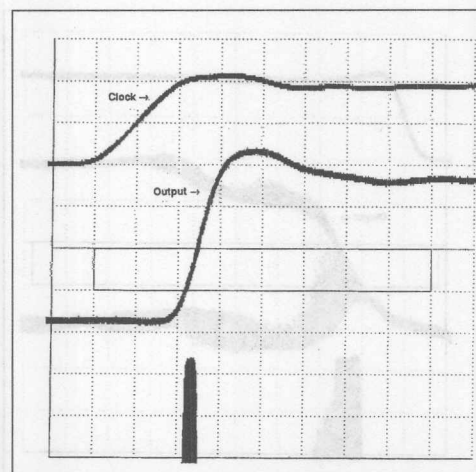
2ns/div

Plot 9. Normal GAL16V8B-7 Transition

Part #	Manufacturer	Δt_0 (ns)	t_{Z} (fms)
T16V8B-7	TI	1.5	1.5

2V/div

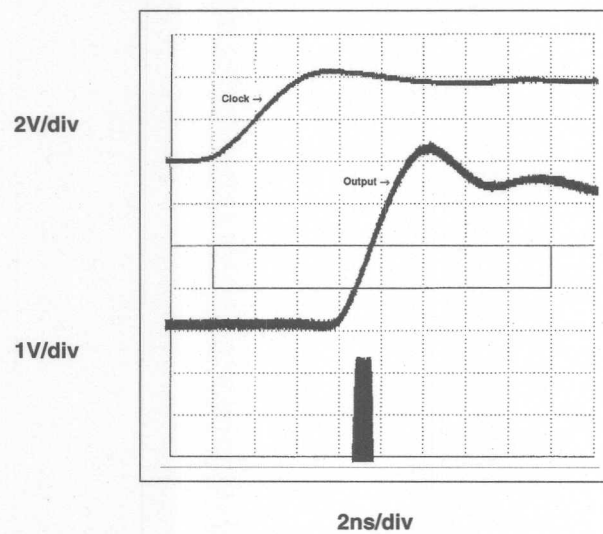
1V/div



2ns/div

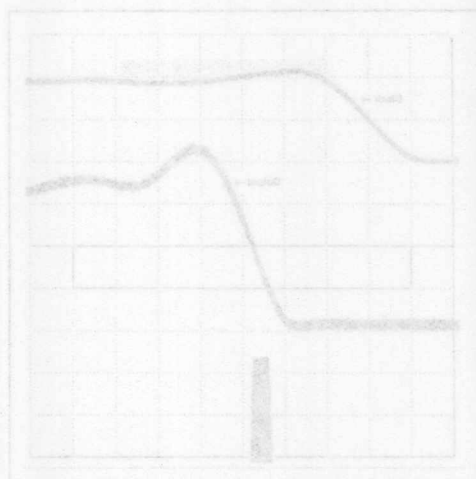
Plot 10. Normal PAL16R8-7 Transition

Part #	Manufacturer	Δt_0 (ns)	t_{Z} (fms)
SHYAS74	TI	2.4	3.2



Plot 11. Normal SN74AS74 Transition

Notes



24115

14115

Plot 11. Normal SNTASTM Transition

24115

END

Latch-up Protection

Introduction

Lattice programmable devices are manufactured using a high-performance E²CMOS process. CMOS processing provides maximum AC performance with minimal power consumption. A drawback common to all CMOS technologies is the potentially destructive phenomenon known as latch-up.

This brief defines latch-up, how it manifests itself, and what techniques have been used to control it. Also described are three device features, employed in Lattice devices constructed with E²CMOS technology (up to and including UltraMOS IV), that prevent the occurrence of latch-up.

Latch-up is a destructive bipolar device action that can potentially occur in any CMOS processed device. It is characterized by extreme runaway supply current and consequential smoking plastic packages. Latch-up is peculiar to CMOS technology, which integrates both P and N channel transistors on one chip.

In the doping profile of a CMOS inverter, parasitic bipolar (PNPN) silicon-controlled-rectifier (SCR) structures are formed. Figure 1 shows the process cross section of a

CMOS inverter, as well as the bipolar components to the parasitic SCR structure. In steady-state conditions, the SCR structure remains off. Destruction results when stray current injects into the base of either Q₁ or Q₂ (see figure 1). The current is amplified with regenerative feedback (assuming that the beta product of Q₁ and Q₂ is greater than unity), driving both Q₁ and Q₂ into saturation and effectively turning on the SCR structure between the device supply and ground. With the parasitic SCR on, the CMOS inverter quickly becomes a nonrecoverable short circuit; metal trace lines melt and the device becomes permanently damaged.

Causes of Latch-Up

It has been explained that parasitic bipolar SCR structures are inherent in CMOS processing. If triggered, the SCR forms a very low-impedance path from the device supply to the substrate, resulting in the destructive event. Two conditions are necessary for the SCR to turn on: The beta product of Q₁ and Q₂ must be greater than unity, which, although minimized, is usually the case; and a trigger current must be present. The cause of latch-up is best understood by examining the mechanisms that produce the initial injection current to trigger the SCR

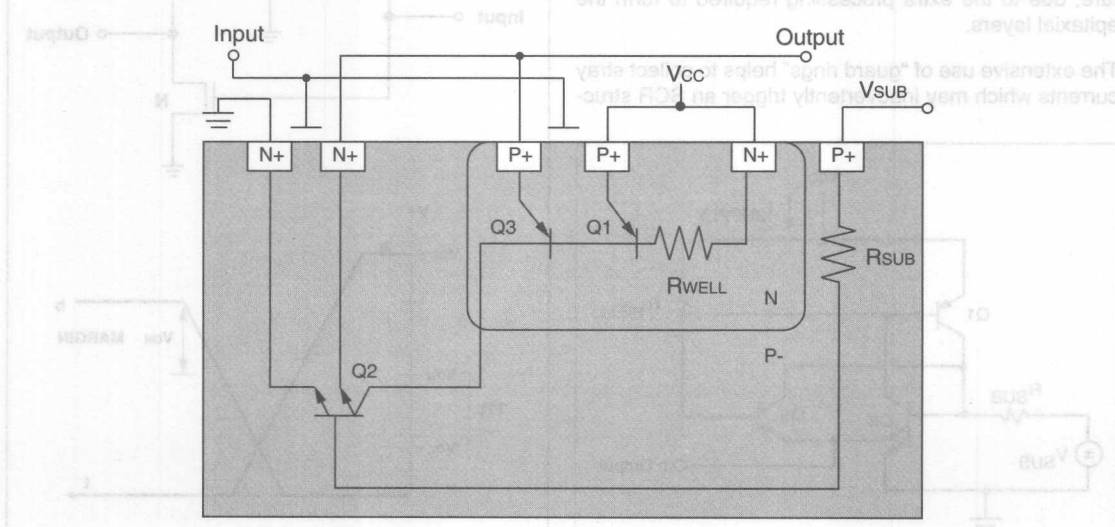


Figure 1. CMOS Inverter Cross-Section

Latch-up Protection

network. Figure 2 is a schematic of the parasitic bipolar network present in a CMOS inverter, where node "b" is the inverter output. It can be seen that two events might trigger latch-up: 1) the inverter output could overshoot the device supply, thereby turning on Q_3 and injecting current directly into the base of Q_2 ; and 2) the inverter output could undershoot the device ground, turning on Q_2 immediately. However, a third condition could also trigger latch-up; if the supply voltage to the P+ diffusion was to rise more quickly than the N-well bias, Q_1 could turn on. Within the device circuitry, overshoot and undershoot can be controlled by design. A problem area exists at the device inputs, outputs and I/Os because external conditions are not always perfect. Powering up can also be a potential problem because of unknown bias conditions that may arise.

With CMOS processing the possibility of latch-up is always present. The major causes of latch-up are understood and it is clear that if CMOS is to be used, solutions to latch-up will have to be created. As the technology evolves, solutions to latch-up are becoming more creative. Two of the more straightforward solutions are presented here.

One direct way to reduce the threat of latch-up is to inhibit Q_2 (figure 1) from turning on. This has been accomplished by grounding the substrate and reducing the magnitude of R_{SUB} through the use of wafers with a highly conductive epitaxial layer. While the technique is successful, the wafers are more expensive to manufacture, due to the extra processing required to form the epitaxial layers.

The extensive use of "guard rings" helps to collect stray currents which may inadvertently trigger an SCR struc-

ture. A disadvantage to heavy use of guard rings is the constraints placed on circuit design and topological layout, and the resulting increase in die size and cost.

The Latch-Lock™ Approach

The intent of the GAL family was to implement cost-effective solutions to each major cause of latch-up. The goal was met through three device features.

The most susceptible areas for latch-up are the device inputs, outputs and I/Os. Extreme externally applied voltages may cause a P+N junction to forward-bias, leading to latch-up. The inputs, by design, are safe; but outputs and I/Os present a danger.

To prevent latch-up by large positive swings on the device outputs or I/O pins, NMOS output drivers were

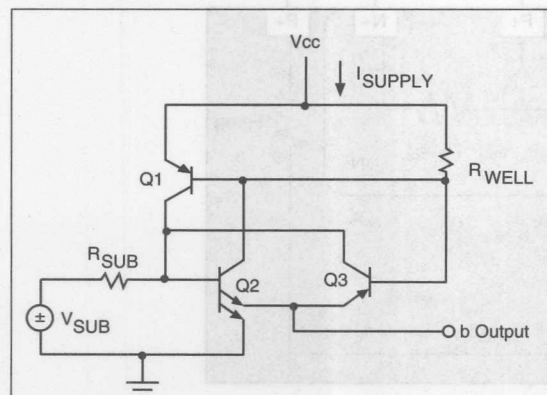


Figure 2. Parasitic SCR Schematic

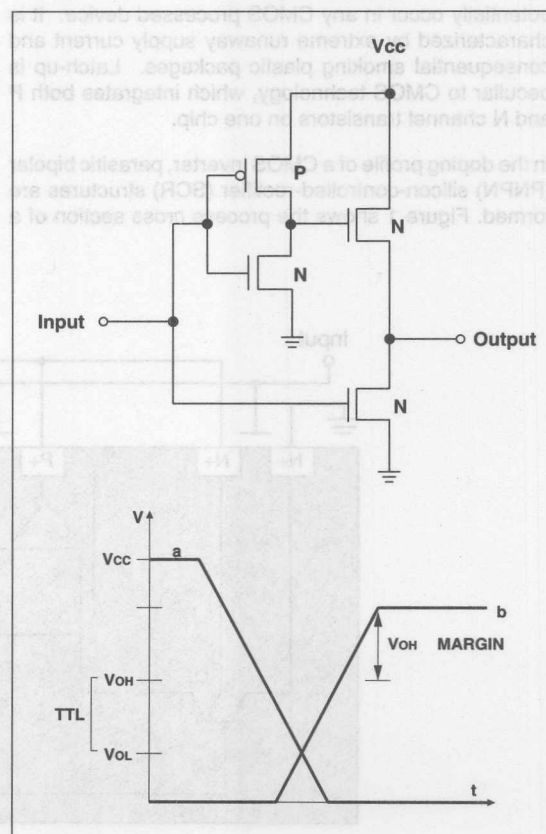


Figure 3. NMOS Output Driver

Latch-up Protection

used. This eliminates the possibility of turning on Q_3 (figure 2) with an output bias in excess of the device supply voltage. Figure 3 contains the effective NMOS output driver and its switching characteristics. Note that the output does not fully reach the supply voltage, but still provides adequate V_{OH} margin for TTL compatibility.

To prevent negative swings on device output and I/O pins from forward-biasing the base-emitter junction of Q_2 , a substrate-bias generator was employed. By producing a V_{sub} of approximately -2.5v, undershoot margin is increased to about -3V.

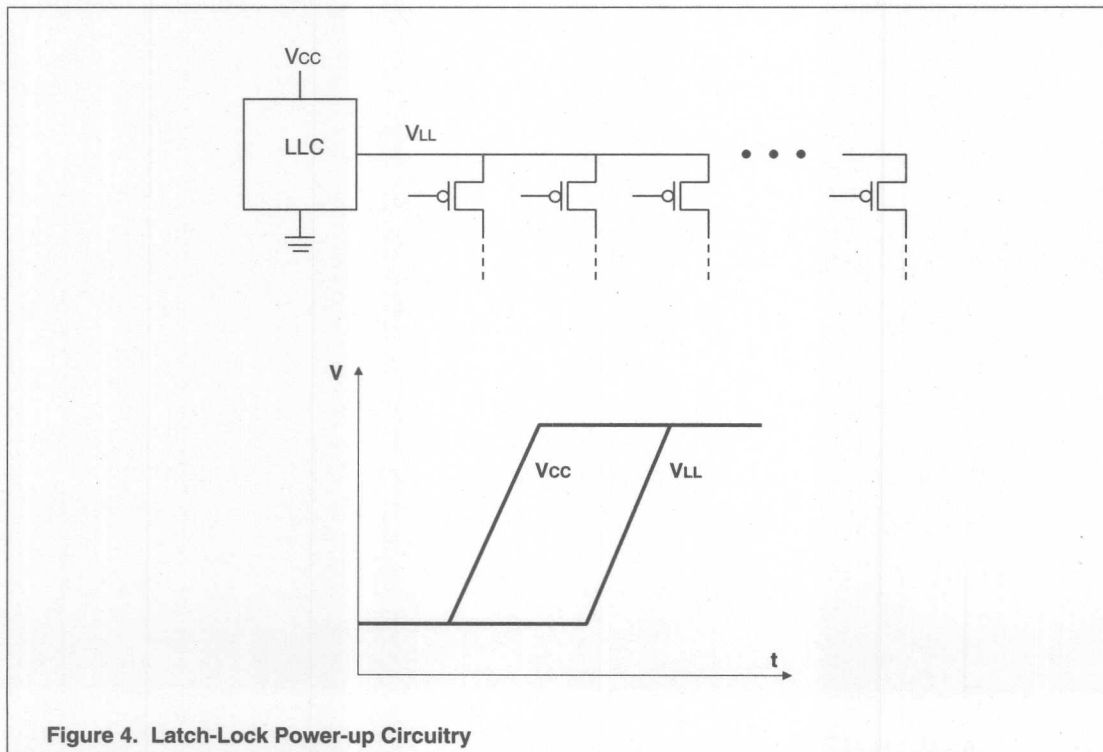
To insure that no undesired bias conditions occur with P+ diffusions, Lattice Semiconductor has developed proprietary Latch-Lock power-up circuitry, illustrated in Figure 4. In short, the drain of all P channel devices normally connected to the device supply is now connected to an alternate supply that powers up after the device N-wells have been biased and the substrate has reached its negative clamp value. This prevents any hazardous bias conditions from developing in the power-up sequence. After power-up is complete, the Latch-Lock circuitry becomes dormant until a full power-down has occurred;

this eliminates the chance of an unwanted P channel power-down during device operation.

To determine the amount of latch-up immunity achieved with the three device features utilized in Lattice devices, an intensive investigation was carried out. Each step was conducted at 25° and 100°C; inputs, outputs, and I/Os were sequentially forced to -8V and +12V while the device underwent fast and slow power-ups; devices were repeatedly "hot socket" switched with up to 7.0V.

Even under the extreme conditions specified, no instance of latch-up occurred. In an attempt to provoke latch-up, $\pm 50\text{mA}$ was forced into each output and I/O pin. The device output drivers were damaged in the battle, and still latch-up was not induced.

Based on the data, it is evident that Lattice devices based on the Latch-Lock technology are completely immune to latch-up, even when subjected to a wide variety of extreme conditions, including current at inputs, outputs, and I/Os, power-supply rise time, hot-socket power-up and temperature.



Notes

This eliminates the chance of an unwanted P channel power-down during device operation.

To determine the amount of latch-up immunity achieved with the three device features utilized in Lattice devices, an intensive investigation was carried out. Each step was conducted at 25°C and 100°C; inputs, outputs, and I/Os were sequentially forced to -8V and +12V while the device underwent fast and slow power-ups; devices were repeatedly "hot socket" switched with up to 7.0V.

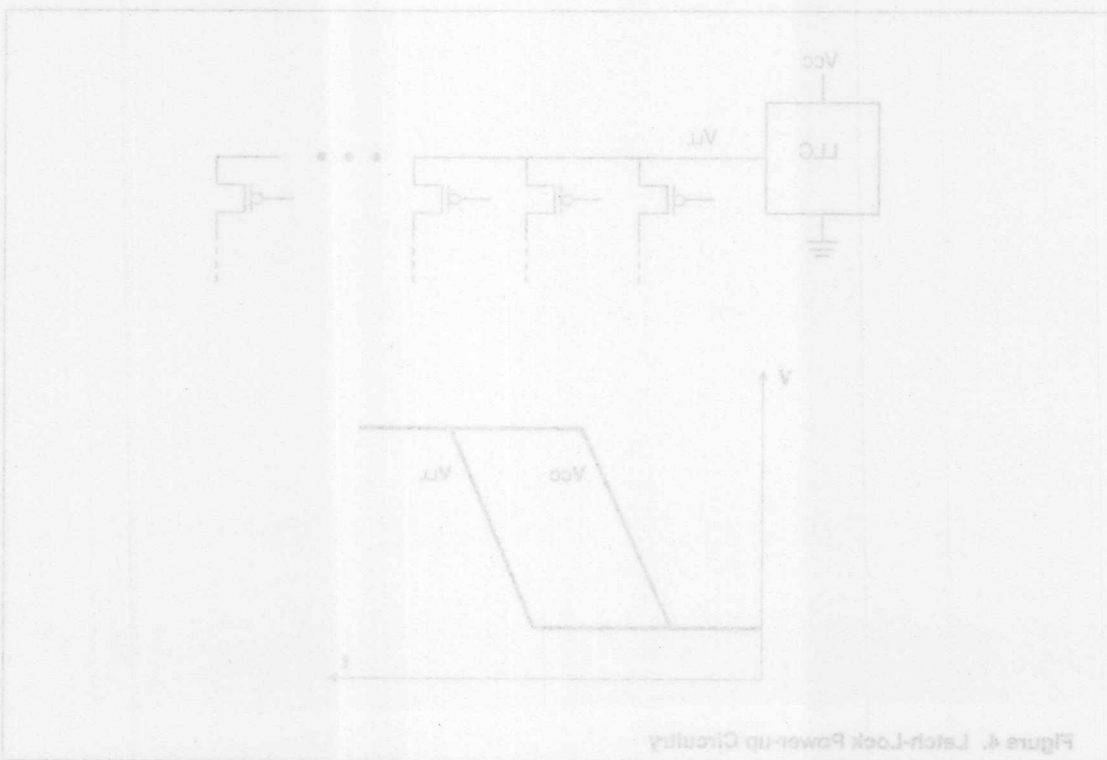
Even under the extreme conditions specified, no instance of latch-up occurred. In an attempt to provide latch-up, a 50mA was forced into each output and I/O pin. The device output drivers were damaged in the battle, and still latch-up was not induced.

Based on the data, it is evident that Lattice devices based on the Latch-Lock technology are completely immune to latch-up, even when subjected to a wide variety of extreme conditions, including current at inputs, outputs, and I/Os; power-supply rise time, hot-socket power-up and temperature.

used. This eliminates the possibility of turning on Q_2 (figure 2) with an output bias in excess of the device supply voltage. Figure 3 contains the effective NMOS output driver and its switching characteristics. Note that the output does not fully reach the supply voltage, but still provides adequate V_{OH} margin for TTL compatibility.

To prevent negative swings on device output and I/O pins from forward-biasing the base-emitter junction of Q_2 , a substrate-bias generator was employed. By producing a V_{sub} of approximately -5.5V, understroke margin is increased to about -3V.

To insure that no undesired bias conditions occur with P+ diffusion, Lattice Semiconductor has developed proprietary Latch-Lock power-up circuitry, illustrated in Figure 4. In short, the drain of all P channel devices normally connected to the device supply is now connected to an alternate supply that powers up after the device I/Os have been biased and the substrate has reached its negative clamp value. This prevents any hazardous bias conditions from developing in the power-up sequence. After power-up is complete, the Latch-Lock circuitry becomes dormant until a full power-down has occurred.



Section 1: Introduction

Section 2: ispLSI and pLSI Architecture Overview

Section 3: ispLSI and pLSI Development Tools

Section 4: ispLSI and pLSI Application Notes

Section 5: GAL Architecture Overview

Section 6: GAL Development Tools

Section 7: GAL Application Notes

Section 8: In-System Programmable Generic Digital Switch (ispGDS)

Section 9: Design Techniques

Section 10: Article Reprints

Selecting the Best Device for In-System Programmability	10-1
Enhanced E ² PLDs Hit Speed and Density Highs	10-7
Complex State Machine Design with Complex PLDs	10-11
Avoid the Pitfalls of Hi Speed Logic Design	10-16
PLD-Design Methods Migrate Existing Designs to High-Capacity Devices	10-23
In-System Programmable Logic in High Volume Manufacturing	10-30
A Token Ring Network Adapter Card	10-37
A Decision Process Used for FPGA Selection in	
Digital Signal Processing for Fiber Optic Sensors	10-44
Learn the Fundamentals of Digital Filter Design	10-51
State Machine Design for High Speed PowerPC RISC Microprocessor Systems	10-59
Applying In-System Reprogrammability in a REFLECTIVE MEMORY Bus Controller	10-66
PLD Usage Generalizes HDTV Frame Buffer Interface	10-73

Section 11: Technology, Quality, and Reliability Overview

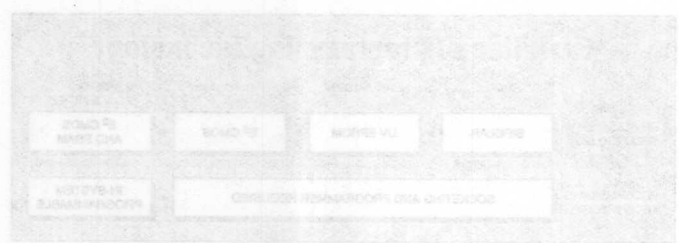
Section 12: General Section

Selecting the best device for in-system programmability

Familiarity with the technologies as well as the benefits and drawbacks involved with each will help you choose the in-system-programmable device that best fits your needs.

Selecting the Best Device for In-System Programmability

This article is reprinted from *Computer Design's ASIC Design*—December 1993.



As PLDs are a subset of the programmable logic device (PLD) family, they are often used to implement logic functions that are not complex enough to require a full ASIC. However, they are also used to implement logic functions that are too complex for a full ASIC.

PLDs are a subset of the programmable logic device (PLD) family, they are often used to implement logic functions that are not complex enough to require a full ASIC. However, they are also used to implement logic functions that are too complex for a full ASIC.

As a designer, you need to be familiar with the technologies as well as the benefits and drawbacks involved with each. This will help you choose the in-system-programmable device that best fits your needs. The first PLD programming technology, only dating from the early days of the 1970s, was the PAL (Programmable Array Logic). PALs were designed to be used in a single place of hardware, but they could be programmed for two different products. The ability to design one product for multiple uses is perhaps the most important feature of PLDs. This is why PLDs are so popular in the design of hardware. They can be used to implement logic functions that are not complex enough to require a full ASIC. However, they are also used to implement logic functions that are too complex for a full ASIC.

Richard Mitchell
Senior product planner,
high-density devices,
Lattice Semiconductor
Hillsboro, OR

Selecting the best device for in-system programmability

Familiarity with the technologies, as well as the benefits and drawbacks involved with each, will help you choose the in-system-programmable complex PLD or FPGA that best suits your design needs.

In developing an HDTV interface, David Harper, a senior design engineer at Convex Computer, faced a problem very common in leading-edge electronics companies: The industry standards were not yet complete, but product development had to go on. His interface board had to connect a Convex I/O channel to the emerging industry-standard frame-buffer format in the infant HDTV electronics market. But two different frame-buffer formats were contending for industry leadership, and it wasn't clear which format would emerge as the dominant standard.

Harper needed to design a system that could accommodate either one, or possibly both, frame-buffer standards. By populating the interface circuit board with several Lattice ispLSI high-density PLDs, Harper designed a product that could be configured to conform to either frame-buffer specification after the logic devices were soldered onto the circuit board. This in-system programmability led to a product with a wider addressable market, as well as lowering design and production costs for Convex, because a single piece of hardware could be programmed for two different products.

The ability to design one product for multiple uses is perhaps the most

exotic benefit of in-system-programmable technology, but it isn't the only one. Designing in-system-programmable devices into a product can improve productivity and reduce costs across the life cycle of a product: engineering, production, maintenance, and in-field upgrades.

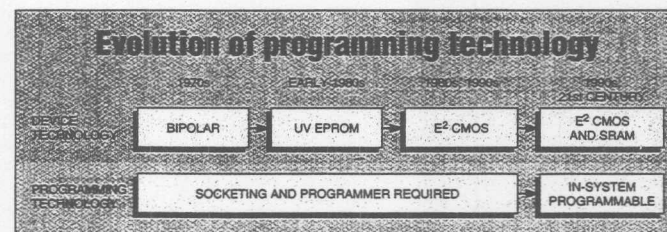
Why in-system programmability?

The first PLD programming technology, dating from the early days of the

Furthermore, manufacturing personnel had to execute a programming step in product manufacturing prior to assembly.

As programmable devices now hurdle the 10,000-gate barrier, however, they incorporate many more I/O pins and are manufactured in very fine pin-pitch plastic quad flat packs and thin quad flat packs. As a result, they're increasingly delicate and intolerant to the manual programming techniques previously used with older packages. Moreover, sockets can reduce the signal integrity of the programmable device and introduce handling and inventory steps that often compromise end-product quality as well as add extra cost.

With an in-system programmable device, on the other hand, the programming pattern can be changed at any time by applying signals to the programming pins of the in-system programmable device. This programming can even be performed after the device is soldered onto the board if the engineer has made an accommodation for the programming interface. This very simple change in the programming step means that design-



As PLDs have migrated from bipolar one-time-programmable technology to UV EPROM and E²CMOS, engineers have had to maintain device sockets and a separate programming step in manufacturing. But in-system-programmable technology does away with all that.

bipolar PAL, programmed tiny fuses on a PLD using a standalone device programmer. As PLDs migrated through subsequent technologies—notably UV EPROM and E²CMOS—engineers continued to rely upon a standalone programming step to load the logic into each device. As a result, engineers would add sockets for the PLDs to their circuit boards in case design changes required new PLDs to be programmed and inserted.

ers don't need to add sockets to have reprogrammability. Nor is a programmer needed. Moreover, in-system programmability makes possible a new way of organizing and executing product development for higher productivity and lower costs.

Selection criteria to consider

To select an in-system-programmable device, you must consider several factors before committing large finan-

Richard Mitchell
Senior product planner,
high-density devices,
Lattice Semiconductor
Hillsboro, OR

cial and engineering resources to a particular technology or architecture. The most fundamental criterion is process technology.

Today's high-density, programmable-logic market offers four basic storage technologies: antifuse, UV EPROM, SRAM and E²CMOS. Although most devices use one of these four technologies exclusively, a few offer a hybrid approach, using SRAM for the logic and EPROM for power-off storage. Of the four basic

configuration when system power is turned off. Every time the system is powered up, the programming pattern must be transferred from an external memory device into the SRAM in-system-programmable devices. This creates two key difficulties. First, the SRAM-based devices require a power-up delay to allow time to load the data from the EPROM device into the SRAM logic cells. Second, additional chips and valuable board real estate are required. Some

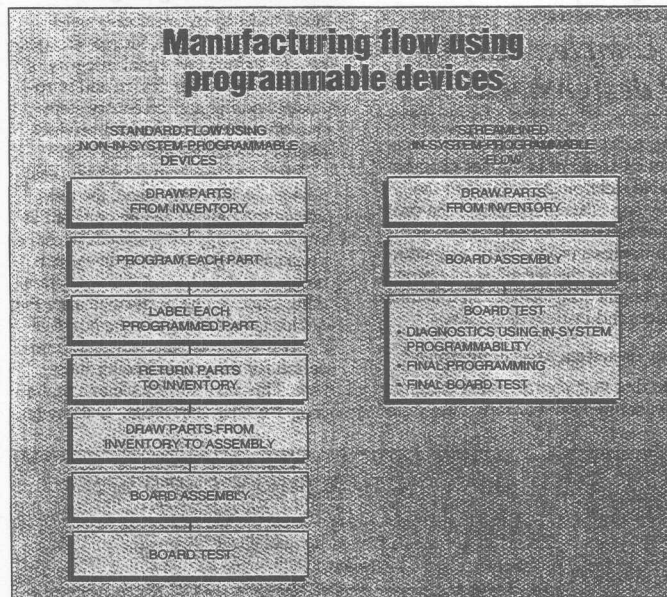
no need for a memory chip to load the logic at each power up, the E²CMOS solution requires no support circuitry, saving critical real estate. E²CMOS devices may be reprogrammed up to 1,000 times, which is sufficient for virtually all in-system-programmable applications except those requiring dynamic or continuous reprogramming.

After device technology, you need to consider programming requirements. Some in-system-programmable devices require a special 12 to 14-V supply to program the device. A board with such devices must incorporate extra control circuitry and must provide this special voltage with an extra power supply such as a dc-to-dc converter. These requirements add up to lost board space with lower reliability, higher power consumption and, ultimately, higher design costs. Other in-system-programmable devices in both SRAM and E²CMOS technologies use a standard 5-V logic supply voltage for programming and reprogramming. This key feature will ultimately lower system design costs and power consumption.

Programming interface options

The next consideration is the simplicity of the programming interface. A simple programming interface conserves board real estate and minimizes layout complexity and design costs compared to complex connectors or programming pad configurations. The key distinction you should consider is whether the interface is serial or parallel. Parallel programming options require extra layout resources for running interface signals across the board. These extra resources inevitably lead to higher design costs. In-system-programmable devices with serial interfaces usually require far fewer interface signals, making them more reliable and cost-effective, as well as easier to design into the system. Serial-interface in-system-programmable devices are available in both SRAM and E²CMOS technologies.

The final consideration concerns the use of in-system-programmable devices as test resources. If test capabilities are a critical design consideration, you should look for devices with interfaces compatible with the IEEE 1149.1 boundary-scan standard.



Using in-system-programmable devices, companies can bypass separate programming, marking, and inventory steps, thereby simplifying the manufacturing flow.

technologies, only SRAM and E²CMOS offer in-system programmability. Of the devices manufactured using these two technologies, not all incorporate in-system-programmable interfaces.

The significant advantage of SRAM devices is that they may be reprogrammed on the fly an almost unlimited number of times. This feature makes SRAM a desirable technology for applications that require continual updates or dynamic-reprogramming capability.

The main drawback of SRAM-based devices, on the other hand, is their volatility. SRAM in-system-programmable devices lose their logic

manufacturers have attempted to eliminate the effect of the second limitation by including the EPROM as part of the device. Although these drawbacks do not always present a serious problem, they eliminate SRAM in-system-programmable devices from consideration in applications requiring fully functional logic at power-up. Examples include a memory decoder for a CPU that must be operational at CPU power-up and applications where board real estate carries a particularly high premium.

The second key in-system-programmable technology, E²CMOS, is non-volatile and electrically erasable within milliseconds. Because there is

In the optimal boundary-scan, in-system-programmable solution, the in-system-programmable and boundary-scan signals share the same dedicated pins. This enables a single interface and the use of identical pins to implement both board test and device reconfiguration.

The need for boundary scan is becoming ever more evident as more components per square inch are packed onto every board. Boundary scan significantly increases test coverage for design errors at the board-test level before they become expensive system-level test problems. Using an in-system-programmable device that isn't compliant with boundary scan offsets this advantage, increasing the risk of higher product-failure rates due to poor testability.

Candidate applications

As systems become denser and more highly integrated, in-system programmability becomes a more critical technology. This is because programmable-logic die is less accessible, making conventional programmer technology inefficient. This situation is especially true for multi-chip modules (MCMs).

In-system programmability offers advantages for many design and system challenges, but not all. This technology is usually not appropriate for extremely high-volume or very cost-sensitive designs because even the minimal additional costs associated with in-system-programmable board overhead may become prohibitive. Products in the early stages of their life cycles, including high-volume applications, are excellent candidates for in-system-programmable devices because they often need numerous changes before they are committed to inflexible ASIC chip sets, for example. A large percentage of new designs can benefit from in-system programmability in production, field upgrades, and generic functionality. And engineers are continually finding more innovative uses for in-system-programmable devices.

More mature products that do not need to be changed during manufacturing or in the field, or only serve one function, however, may not need the benefits that in-system-programmable devices offer because by their very nature they do not require many engineering change orders.

Multi-chip module packaging has perhaps the most critical need for in-system programmability. As demonstrated earlier, in-system programmability streamlines design processes using delicate fine-lead packaging and increasingly dense circuit boards. It also contributes to a more complete test strategy. With the higher integration and intricacies of MCMs, each of these advantages becomes more significant. Conventional non-in-system-programmable technologies require the die to be removed from the module, a practice that often damages the MCM. With in-system programmability this problem is nonexistent.

Test is another point of contention with MCMs as the modules cannot be tested with conventional techniques. Using an in-system-programmable

greatly reduce the design/debug cycle as well as prototype-development time. In a normal design cycle, an engineer generally designs the circuit, has a circuit board built, and begins debugging the board. In most circuits, the engineer corrects the inevitable errors and implements specification and design changes by physically removing socketed PLDs and inserting updated versions. He may have to modify the circuit board's traces and layout to accommodate resulting design and pinout changes.

When using in-system-programmable devices, design changes that previously took a half day of cuts and jumpers on the prototype board take just minutes. You make changes to the logic equations within the PLD and send the new programming to



Programmable logic parts with in-system programming capabilities continue to grow in popularity as more board designs adopt PQFP, TQFP and other fine-lead packages. Frank Morris, Valerie Young and Brian Reilly implemented the logic on NEC America's digital loop carrier board with four in-system programmable devices from Lattice to ensure lead conformance and overall product quality.

design strategy that incorporates boundary scan will enable almost full testability of an MCM. The designer may fully develop the MCM package and attach the die before the design is fully tested. When errors are found, the design can be reprogrammed inside the MCM with a very short turn-around time.

Optimized development cycle

The use of in-system-programmable devices in the engineering lab can

the PLD on the circuit board through the programming interface. This quick design-turnaround time can save you days to weeks within a development schedule.

Since in-system-programmable devices eliminate the need for prototyping sockets, there's no need to redesign the prototype board for production. Because the prototype and production boards may be identical, their capacitance and inductance, and, therefore, their ac performance,

may also be identical. This eliminates unpleasant surprises in product performance when the first production units are manufactured.

The opening Convex example points out another benefit: A single circuit board may be designed for multiple uses. This technique is also known as reconfigurable hardware. With reconfigurable hardware, an engineer can design and build a product and then reconfigure it to accommodate any number of industry standards or product features. A

simplified manufacturing flow, leading to higher quality and more accurate prototypes. First, as thin quad flat packs, plastic quad flat packs and other fragile chip packaging gain in popularity, manufacturing and assembly of circuit boards with sockets can become a serious quality-control problem. The additional handling steps for programming often leads to bent pins and lower part-utilization rates. With an in-system-programmable design strategy, inventory hassles are significantly reduced because the

programming pins, a field-service technician or even a customer can upgrade a product's hardware as easily as software upgrades are distributed today.

System reliability issues

Socketing of programmable devices can be a consistent source of system-reliability problems. By removing sockets from the circuit board and soldering an in-system-programmable device directly onto the board, the signal integrity of the leads isn't compromised by the socket connections. In addition, properly soldered joints are much more reliable than socket connections. These benefits result in greater system performance and overall reliability in terms of MTBF.

The use of in-system-programmable devices also lets the test engineer develop more flexible circuit-board-test procedures. For example, a test engineer can program in-system-programmable devices to interconnect the circuit-board traces and so achieve nearly full fault coverage of those traces. He or she can then quickly reconfigure the in-system-programmable devices to generate test signals for exercising dedicated logic devices on the board. By doing so, test engineers can significantly enhance the fault coverage of the board and the speed of tests. After board test, the test engineer can program the final logic patterns into the in-system-programmable devices.

Design engineers can use in-system-programmable devices to design boards with programmable configuration options, instead of dip switches or component swapping. This multiple-configuration approach can greatly improve system-level performance and reliability by reducing device counts, eliminating the need for sockets and improving testability.

The Lattice isp solution

After considering in-system programmable benefits, you can evaluate various device families for features that match their system requirements. Lattice Semiconductor, for example, fields three in-system-programmable LSI (ispLSI) device families based on the company's proprietary E²CMOS technology: the flagship ispLSI 1000 family and the recently announced ispLSI 2000 and ispLSI 3000 families.

Programmable Logic Technology Comparison

	E ² CMOS	SRAM	Antifuse	UV EPROM
Non-volatile	Yes	No	Yes	Yes
Single-chip solution	Yes	No	Yes	Yes
Reprogrammability	Yes	Yes	No	Yes (slow)
In-system programmability	Yes	Yes	No	No
Program time	Fast	Fast	Slow	Fast
Erasure time	Fast	Fast	N/A (one-time programmable)	Slow
Testability	Full	Full	Limited	Limited
Yield	100%	100%	93-97%	98-99%

generic PC card could be customized, for instance, to work with different network protocols.

Optimized production flow

Using typical PLDs, the production flow consists of taking blank parts from inventory, programming and marking them, sending each part back to inventory with a specific part number, then pulling the appropriate part number as needed to assemble the production cards. Programming a high-pin-count PLD is problematic because its fine pin pitch makes it incompatible with automatic programmer handlers. Consequently, production personnel must program all conventional high-density PLD devices manually. An operator has to place and remove each device from the socket on the programmer, a task that's very difficult to accomplish without severely bending the fine leads or destroying lead coplanarity.

A product using in-system-programmable devices enjoys a much

parts go directly from the receiving dock to placement on the printed circuit board, eliminating the stand-alone programming and mark operation entirely. In addition, multiple, blank in-system-programmable devices can be loaded into auto-insertion equipment and placed directly onto the board without sockets and without regard for which pattern goes into a particular board location. During final circuit-board test, the individual logic patterns are programmed into each device using the board-test station via the simple in-system-programmable programming interface.

Many products also require field upgrades to maintain their accuracy or to update their functionality. Instrumentation equipment is a good example because it often requires recalibration over a period of months or years to maintain accuracy and precision. With in-system-programmable parts embedded in the system and an appropriate interface to the

The ispLSI 1000 family, introduced in 1991, was the first available E²CMOS in-system-programmable solution on the market to offer 2,000 to 8,000 gates and up to 110-MHz speed. The ispLSI 2000 family expanded upon the ispLSI 1000 family architecture to deliver speeds of up to 135 MHz. The ispLSI 3000 family offers device densities of up to 14,000 gates with 110-MHz speed and IEEE 1149.1 boundary-scan test capabilities.

The Lattice ispLSI devices follow a simple reprogramming scheme. Five pins are dedicated to in-system programming: serial data in (SDI),

The ability to design one product for multiple uses is perhaps the most exotic benefit of in-system-programmable technology, but it isn't the only one. Designing in-system-programmable devices into a product can improve productivity and reduce costs across the life cycle of a product.

serial data out (SDO), mode control (Mode), serial clock (SCLK) and isp enable (ispEN). During the reprogramming operation, ispEN is asserted low, the four remaining ispLSI pins become active, and all other output pins become three-stated to prevent any bus contention during the reprogramming cycle. The programming of the device is then controlled by an internal state machine that's operated by using the SDI and Mode pins. You would use the design software provided by Lattice on a workstation or PC to serially load a 5-bit command into the device, followed by the design file in JEDEC format, all using a 5-V reprogramming voltage. Lattice also offers a software routine called ispCODE which gives you pre-written working C routines that can be incorporated in a system processor as part of the working system code.

Lattice's ispLSI 3000 family of devices share the isp programming signals with the standard boundary-scan signals, enabling the same interface to do both board test and logic reconfiguration.

An isp design example

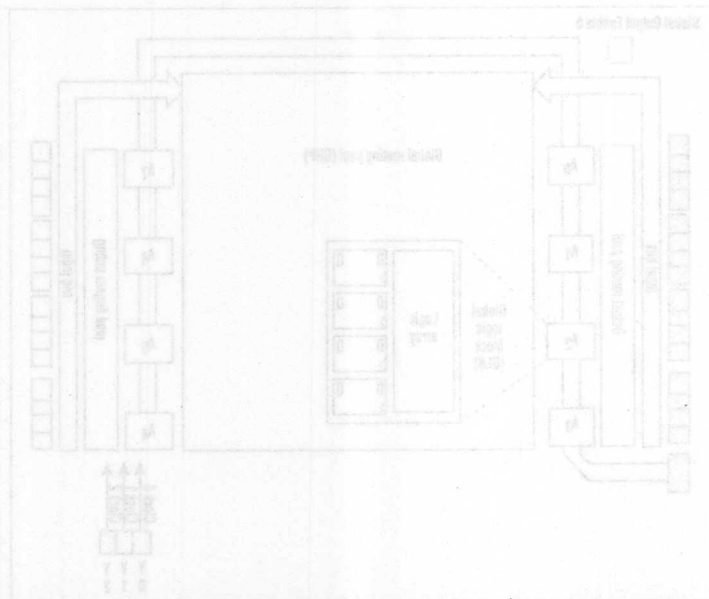
Brian Reilly of NEC America (Hillsboro, OR) found an application that may not have been completed without ispLSI devices. Reilly's task was to design new circuit boards for an NEC digital loop carrier that accepts 96 phone pairs and digitally compresses them down to eight pairs. The boards were to be part of the system's common control unit and consisted of a 68020-based control CPU board and a custom high-capacity, serial-interface board. NEC encountered a set of engineering problems: trying to successfully implement the functional requirements which included the need for non-volatility, maximizing the amount of logic in the smallest amount of board real estate, minimizing board- and system-test costs, maintaining high product reliability and minimizing board rework caused by engineering change orders.

While all these constraints pointed to in-system programmability, the criterion that made in-system programmability unavoidable was the need for NEC to start building the hardware before the logic was fully designed. By implementing the design with Lattice ispLSI devices, NEC was able to meet a very tight product-development cycle, one which would have been impossible without an in-system-programmable strategy. Board layout was finished, and assembly took place weeks before the final logic was completed and implemented into the devices. Looking back on the experience, Reilly says, "The ability to change the logic on the board really saved our bacon." □

ENHANCED E²PLDS HIT SPEED AND DENSITY HIGHS

Enhanced E²PLDs Hit Speed and Density Higgs

This article is reprinted from *Electronic Design* — October 14, 1993.



THE OVERALL ARCHITECTURE of the enhanced 3000 and 3000 series of enhanced programmable logic devices is similar to that of the original 1600 series. As the result of the research developed by Lattice Semiconductor, a global control logic (GCL) surrounding the GCL and the macrocell, which are in turn surrounded by the original routing paths and the I/O pins.

ENHANCED E²PLDs HIT SPEED AND DENSITY HIGHS

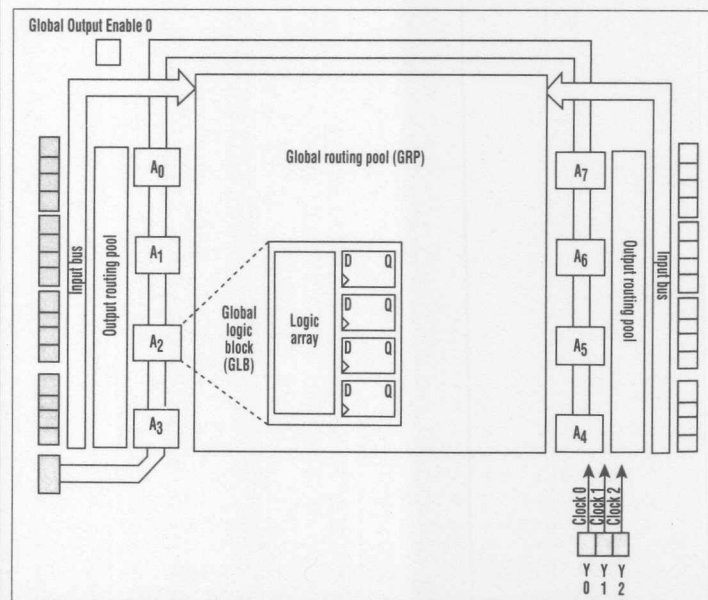
DAVE BURSKY

TWO COMPLEX
PLD FAMILIES
LET DESIGNERS
RUN SYSTEMS
AT 135 MHz
OR TRIM CHIP
COUNT WITH
14-KGATE
PROGRAMMABLE
LOGIC ARRAYS.

Increasing complexity in programmable logic devices presents designers with two key challenges: How to maintain the short propagation delays found in the low-complexity devices while achieving the high functionality possible with gate-array alternatives. To address both of these areas, two enhanced families of high-density programmable logic were developed by Lattice Semiconductor. Both families will use the company's electrically-erasable programmable logic devices (E²PLDs). Each will come in Lattice's standard programmable LSI (pLSI) and in its unique in-system programmable LSI (ispLSI) form.

The first of the two families, the 2000 series, offers the shortest propagation delays for complex PLDs. The other family, the 3000 series, focuses on gate density, I/O count, and enhanced testability through the use of an IEEE 1149.1-compatible boundary-scan test port. Each family will initially have three density options. Both will join the company's original pLSI and ispLSI 1000 families of E²CMOS high-density PLDs.

The 2000 family, with operating frequencies up to 135 MHz and pin-to-pin logic delays of as little as 7.5 ns, suits many of the applications that currently



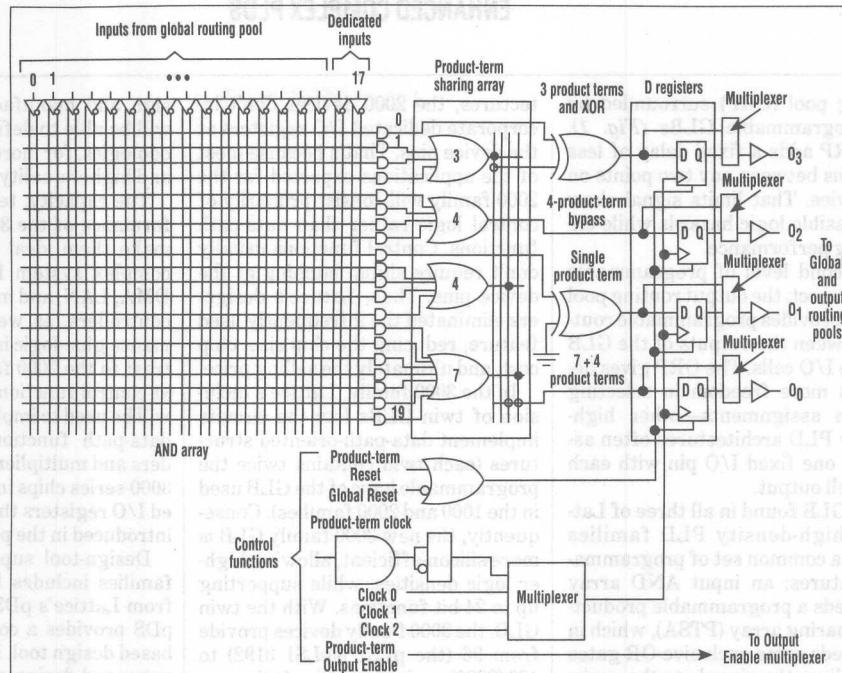
1. THE OVERALL ARCHITECTURE of the just-released 2000 and 3000 series of enhanced programmable logic circuits is similar to that of the original 1000 series. At the heart of the circuits, developed by Lattice Semiconductor, is a global routing pool (GRP). Surrounding the GRP are the macrocells, which are in turn are surrounded by the output routing pools and the I/O pads.

employ high-speed but low-density PLDs. This series also widens the logic density and I/O count beyond the levels available in the earlier 1000 family. In part, the high speed was achieved by using the company's 0.65- μ m, UltraMOS V E²CMOS process. Additional speed gains were the result of architectural streamlining.

The family's three density options—the 2032, 2064 and 2096—contain the equivalent of 1000, 2000, and 4000 PLD gates, respectively. Those gates are divided among 8, 16, and 24 generic logic blocks (GLBs), respectively, on each of the three chips. Each GLB possesses 20 product-term inputs and provides four registered or combinatorial outputs. It allows the designer to configure each of the four GLB flip-flops as a JK-, T-, or D-type element, and gives the designer multiple clocks for synchronous or asynchronous applications. Inputs and I/Os total 34 for the 2032, 68 for the 2064, and 102 for the 2096. Package options include PLCC, plastic quad flat pack (PQFP) and thin quad flat pack (TQFP), with pin counts ranging from 44 to 128 leads.

Taking the high road in terms of logic density, I/O counts, and features, the 3000 family will initially offer three densities ranging from 8000 to 14,000 PLD-type gates—the 3192, 3256, and 3320. The pLSI and ispLSI versions will offer top clock speeds and minimal propagation delays that span from 110 MHz/10 ns for the 3192 to 80 MHz/15 ns for the 3320. These chips provide 24, 32, and 40 programmable GLBs, respectively, with inputs and I/O pins totalling 96 for the 3192, 128 for the 3256, and 160 for the 3320.

The 3000-family GLBs are more complex than those in the 2000 series—each 3000-series GLB has 24 inputs from the global routing pool and 8 register outputs. That gives



2. EACH GLOBAL LOGIC BLOCK (GLB) used in the pLSI/ispLSI 2000 series offers a similar level of functionality as the GLBs in Lattice's original pLSI 1000 series. The blocks, which provide 18 inputs, 20 product terms, and 4 register outputs, are well suited for control applications. The more complex macrocell used in the 3000 series packs close to double the functionality and more readily handles data-path type functions.

the series 3000 almost twice the complexity of the 2000 series, considerably improving the flexibility and ability to implement more-complex functions in the GLB. The programmable circuits will be available in 128-, 160-, and 208-lead PQFPs.

The ispLSI version in all three families makes it possible for them to be programmed using only a standard 5-V power supply and a five-wire serial interface. In-system programmability addresses many of the manufacturing concerns voiced by companies using fine-pitch, high-pin-count programmable devices on surface-mount pc boards. By eliminating separate programming steps and socketing operations, ispLSI chips minimize concerns about bent device leads and out-of-specification lead coplanarity due to handling.

Providing short-delay programmable logic to support high-performance system designs has been the exclusive domain of low-density but

fast PLDs. High-speed address decoding and minimal-delay bus-controller functions must typically operate at twice the processor clock frequency. Even the fastest 20- and 24-pin PLDs have been hard-pressed to keep up with the performance requirements of the Pentium, Alpha, and other RISC and CISC CPUs.

The Pentium's 66-MHz clock rate dictates that the support logic operate at speeds of at least 132 MHz. Thanks to Lattice's UltraMOS V process, the 44-pin, 8-GLB pLSI and ispLSI 2032, which operate at 135 MHz, are now the first high-density PLDs to exceed this threshold (the 132-MHz minimum frequency). And because the 2032 integrates the equivalent of 4 to 6 low-density PLDs like PALs into a single package, designers can cut power consumption and board space by up to 6:1.

The 2000 family architecture is similar to that of the earlier 1000 family. It consists of a central global

ENHANCED COMPLEX PLDS

routing pool (GRP) surrounded by the programmable GLBs (Fig. 1). The GRP adds a fixed delay of less than 2 ns between any two points on the device. That limits signal skew and possible logic hazards while enhancing performance.

A second level of programmable interconnect, the output routing pool (ORP), provides programmable routing between the outputs of the GLB and the I/O cells. The ORP gives designers more freedom in selecting the pin assignments—other high-density PLD architectures often associate one fixed I/O pin with each logic-cell output.

The GLB found in all three of Lattice's high-density PLD families shares a common set of programmable features: an input AND array that feeds a programmable product-term sharing array (PTSA), which in turn feeds some exclusive-OR gates that deliver the signals to the registers or directly to the output-routing matrix (Fig. 2). For example, the 2000-family GLB programmable logic array takes 18 inputs from the GRP and generates combinatorial AND-OR functions. The outputs of the OR gates are routed through the PTSA, in which common functions can be efficiently shared and incorporated into multiple GLB output functions. The integral XOR gates make the GLB particularly efficient in handling arithmetic and comparator functions.

For very-high-speed functions, the PTSA can be bypassed to give the fastest input-to-output path. The registers in the GLB can be independently programmed for either JK-, T- or D-operation. In addition, multiple clock options, including product-term clocking, make synchronous circuit implementations easier.

Sticking to its role as a fast, cost-conscious family of devices, Lattice took a "lean and mean" approach with the 2000-family architecture. For example, unlike the original 1000- and the new 3000-family archi-

tectures, the 2000 devices don't incorporate dedicated I/O registers at the device pins. That's because most of the applications expected for the 2000 family will consist primarily of control logic rather than data-path functions. Control functions usually don't require signal latching at the device pins. Thus, Lattice's designers eliminated the infrequently used feature, reducing the chip size, chip cost, and ultimately the selling price.

In the 3000 family, Lattice's inclusion of twin GLBs lets the circuits implement data-path-oriented structures (each twin contains twice the programmable logic of the GLB used in the 1000 and 2000 families). Consequently, the new 3000 family GLB is more silicon-efficient, allowing higher logic densities, while supporting up to 24-bit functions. With the twin GLB, the 3000 family devices provide from 96 (the pLSI/ispLSI 3192) to 480 (3320) registers in one device.

The 3000 family also is enhanced significantly by the integration of dedicated boundary-scan logic into the I/O structure. The 3000-family devices include IEEE Standard 1149.1-compliant JTAG (Joint Test Advisory Group) circuitry, which allows the scan data pattern to be shifted in or out. As part of the JTAG port, each device includes a test-access-port (TAP) controller and associated I/O scan registers. All mandatory 1149.1 public instructions are supported, including Bypass, Sample/Preload and EXtest. The boundary-scan feature enables complex-pc-board and systems designs to exhaustively test overall logic functionality to ensure the highest quality and reliability levels.

The isp programming interface and boundary-scan TAP in the 3000 family use the same set of four pins, selected by an isp/boundary-scan mode-select pin. Clock, Serial Data In, Serial Data Out, and Mode control pins are present in both interfaces. By combining in-system programming and boundary scan, de-

sign and manufacturing engineers will be able to define new test methodologies, for more thorough testing and higher quality.

The capacity, testability, and performance of the 3000 series devices make them ideal for implementing complex system functions such as DMA, LAN, and memory subsystem controllers, as well as high-performance glue-logic integration. In contrast to the 2000 family, which aims at control functions, the 3000 family will be used to implement significant data-path functions, including adders and multipliers. As a result, the 3000-series chips include the dedicated I/O registers that were originally introduced in the pLSI 1000 family.

Design-tool support for the new families includes PC-based support from Lattice's pDS design tool. The pDS provides a complete Windows-based design tool, including Boolean entry and design editing along with automatic logic routing. Lattice currently provides support for popular third-party design environments on the PC and Sun platforms, such as pDS+ Viewlogic, pDS+ ABEL and pDS+ LOG/iC design fitters. Viewlogic's Viewsim timing simulation and Logic Modeling system-level simulation models are also available. Support for Cadence, Mentor Graphics and Synopsys design tools is also planned. □

By Bernard Lough
Senior Applications Engineer
Lattice Semiconductor Corporation

The alternative to scaling product terms with a PAL type architecture is to cascade multiple PAL type architectures to implement the same machine. In this case multiple layers of product term blocks are used.

State Machines have become an integral part of the system design. With the price reduction and

Complex State Machine Design with Complex PLDs

This paper was presented at the 1993 Silicon Valley Personal Computer Design Conference in Santa Clara, California, and appeared in the SVPC '93 Conference Proceedings Volume II.

State Machine Design Requirements

Having enough registers is a paramount concern in designing state machines. Some architectures provide a register in the I/O macrocell that allows the next to "latch" or "register" input signals. This is an ideal feature in bus arbitration/producer-consumer applications. Input registers or latches also facilitate the design of fully synchronous state machines.

The type and quantity of registers contained in a particular device can not be over stressed. The type of register will have a dramatic impact on the number of product terms and internal resources required to implement the state machine.

In the case of architectures that incorporate product steering, typically what occurs is that to satisfy the high product term requirements, product terms are borrowed from adjacent macrocells. The disadvantage to this approach is that in reality the product terms are stolen from the adjacent cell reducing the cell's available and wasting a register.

Some PAL architectures attempt product term sharing by steering clusters of product terms to adjacent outputs, or by having a common pool of product terms that are shared among all the outputs.

By using an FPLA architecture, product term steering may be used, which allows the user to "share" terms among registers. This capability is the result of the programmable OR array.

opposed to a standard PAL architecture which contains a single programmable AND array. and a programmable OR array is optimum as which contains an programmable AND array) in FPLA (Field Programmable Logic Array) design is to be implemented. For these reasons term sharing is an efficient and economical way to support product rich and support product.

common to multiple outputs. terms used within a state machine design output and with a high percentage of the product intensive, with a high percentage of the product

State machines appear in a large amount of control an interfacing functions. State

machine architectures suitable for high speed machine applications. spawned an architecture for high speed state machines. Functions into Programmable Logic State design engineers are incorporating their control

State Machines have become an integral part of the system design. With the price reduction and

Introduction

State Machines have become an integral part of the system design. With the price reduction and fast time to market of Programmable Logic more design engineers are incorporating their control functions into Programmable Logic State Machines. This increase in popularity has spurred an increase in the available PLD architectures suitable for high performance state machine applications.

State machines appear in a large amount of PLD designs, and are at the heart of almost all control an interface/arbitrate functions. State machines are generally very product term intensive, with many product terms required per output and with a high percentage of the product terms used within a state machine design common to multiple outputs.

State machines require PLD architectures that are very product term rich and support product term sharing if an efficient and economical design is to be implemented. For these reasons an FPLA (Field Programmable Logic Array) which contains an programmable AND array and a programmable OR array is optimum as opposed to an standard PAL architecture which contains a single programmable AND array.

By using an FPLA architecture, product term sharing may be used, which allows the user to "share" terms among registers. This capability is the result of the programmable OR array.

Some PAL architectures attempt product term sharing by steering clusters of product terms to adjacent outputs, or by having a common pool of product terms that are shared among all the outputs.

In the case of architectures that incorporate product steering, typically what occurs is that to satisfy the high product term requirements, product terms are borrowed from adjacent macrocells. The disadvantage to this approach is that in reality the product terms are stolen from the adjacent cell rendering the cell unusable and wasting a register.

The alternative to stealing product terms with a PAL type architecture is to cascade macrocells to implement the state machine. In this case multiple layers of product term blocks are ORed together to supply the required number of product terms.

Another approach is to use an architecture which contains a pool of uncommitted product terms which must be cascaded in single product term blocks to provide the total number of product terms needed. The down side to this approach is that each of the "spare" or uncommitted product terms will add an additional delay. With some devices the additional delay may be as much as 15nsec for each additional product term used. This delay is accumulative and results in extremely slow operating speeds and unpredictable delays. It is difficult if not impossible to determine how many product terms are required for a given state of a multiple state state machine, therefore the cumulative delay of the device is not known until after the design is complete. If the state machine is required to operate at 30Mhz or faster than a product term pool architecture is probably not a wise choose.

A FPLA type of architecture is typically the best solution for PLD based state machine designs. The following section will discuss why.

State Machine Design Requirements

Having enough registers is a paramount concern in designing state machines. Some architectures provide a register in the I/O macrocell that allows the user to "latch" or "register" input signals. This is an ideal feature in bus arbitration/sequencer applications. Input registers or latches also facilitate the design of fully synchronous state machines.

The type and quantity of registers contained in a particular device can not be over stressed. The type of register will have a dramatic impact on the number of product terms and internal resources required to implement the state machine.

The first consideration should be which registers are available in hardware. While many manufactures claim that a "D" type flip-flop can emulate a JK, T or SR, the logic required to do so is typically prohibitive due the additional product terms and time penalties since additional feedback is required. Thus, having the right flip-flop in hardware can greatly improve the efficiency and speed of the design. This is why PAL type architectures are ineffective in implementing state machines.

T flip-flops do well in counter applications where state/output bits must toggle upon a transition of the clock. JK or SR flip-flops efficiently implement the "if..then..else" statement in state machine language. The JK flip-flop can also have a T flip-flop function when both J and K inputs are active.

State machines often require a portion of the output registers and state bit registers to remain in the same logic condition during a transition from one state to another, in other words the register must "hold" its value. For example, when state bits S3-S0 change from 1111 state to 1110 state, the only state bit that makes the transition is the S0 state bit. The state bits S3-S1 must hold the logic '1' state. When implementing this state transition with D flip-flops, one must define the conditions for each of the register that must hold the logic condition. As a result the equation is defined as follows.

$S3.D = S3 \& S2 \& S1 \& S0$
 Hold S3 high when $S3-S0 = 1111$
 $S2.D = S3 \& S2 \& S1 \& S0$
 Hold S2 high when $S3-S0 = 1111$
 $S1.D = S3 \& S2 \& S1 \& S0$
 Hold S1 high when $S3-S0 = 1111$

Similarly, when implementing the same state transition with the JK flip-flops, one must only define the state bit that made the transition. State bit S0 transition equation is defined as follows.

$S0.K = S3 \& S2 \& S1 \& S0$
 Reset S0 high when $S3-S0 = 1111$

As a result, the D flip flop implementation takes three product terms where the JK flip-flop implementation only takes one product term to implement the same state transition. The D flip-

flops generally requires extra product term(s) over the JK flip-flops in order to hold its value which result in a less efficient implementation.

Many of the state bits used in a state machine design are imbedded or "buried". If a standard PAL type architecture is used in a state machine design any bits that are feed back into the array will use or occupy an I/O pin. Because of this, extra attention must be paid when defining the state bits. If an Output Logic Macrocells output is feed back into the AND array and cannot be used as the functional output, the I/O pin associated with that macrocell can no longer be used. Therefore device architectures that provide a separate feed back path, allow the OMLC to be buried without losing the I/O pin.

Enhanced DE Flip-Flop

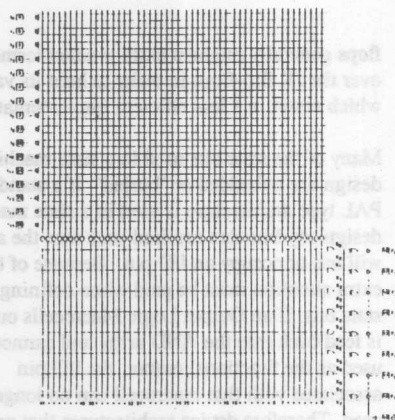
Several FPLA manufactures have developed a DE flip flop. The clock Enable signal of a DE flip-flop can be used to hold the current register value by disabling the clock. This has the advantage that changes on the data signal will not affect at the register's output when the register is "holding" a state bit. Using the same state transition example as above, the equations are defined as follows.

$S0.E = S3 \& S2 \& S1 \& S0$
 Enable S0 clock when $S3-S0 = 1111$

Notice that the DE flip flop has the same efficiency as the JK. Additionally, with an DE type register the "E" term can be used as an asynchronous clock input for the flip-flop. Devices that incorporate DE type registers allow the user to synchronously or asynchronously clock macrocells on a macrocell by macrocell basis.

FPLA Basics

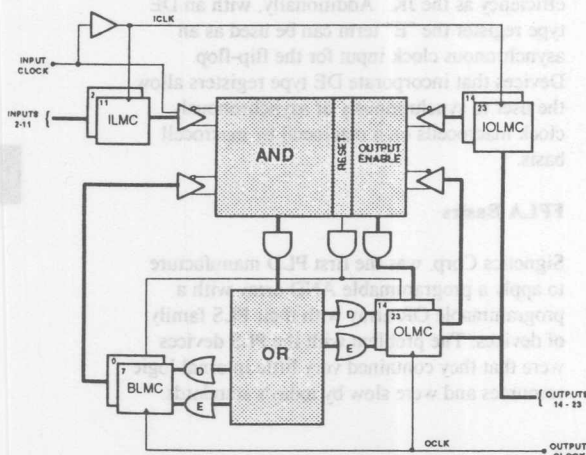
Signetics Corp. was the first PLD manufacture to apply a programmable AND array with a programmable OR array with their PLS family of devices. The problem with the PLS devices were that they contained very little internal logic resources and were slow by today's standards.



PLS

Several manufacturers have taken the original concept of the PLS FPLAs to the next stage by incorporating input capture registers, buried registers, independent I/O and feed back routing channels and DE type registers.

An architecture that incorporates the features required to design an efficient high speed state machine can be found in the GAL6002 device. The GAL6002 is the most complex low density PLD in the world. It should also be noted that the GAL6002 is provided in a 24 pin DIP or 28 pin PLCC package and is input, output and power pin compatible to the industry standard 22V10 PAL architecture.



GAL 6002

The GAL6002 is the most densely populated 24 pin PLD available. The GAL6002 is an FPLA (Field Programmable Logic Array) which has an programmable AND array and a programmable OR array. The 6002 contains a total of thirty eight registers and two synchronous clocks. The device has input macro cells which can be configured as a D type register, a D latch or as a combinatorial input. These input macrocells can be individually configured by the user. The input macrocells feeds into the programmable AND array and then into the programmable OR array. Within the programmable OR array there are eight buried macrocells that feed back into the programmable AND array and ten output macrocells that can either feed back into the AND array or drive the outputs directly.

The DE flip flops within the buried and output macrocells can be configured as a T, D, JK or SR type flip flop. The truth table below shows the input and output conditions for the DE flip flop.

D	E	Q output
0	0	Q
0	1	0
1	0	Q
1	1	1

Table 1. DE Flip-Flop Truth Table

The following paragraphs provides the simple conversions of each of the JK, SR and T registers to the DE register.

SR to DE:

The original SR definition provides the $Q_{OUT.S}$ and $Q_{OUT.R}$ equations. These equations are then converted as:

$$Q_{OUT.D} = Q_{OUT.S}$$

$$Q_{OUT.E} = Q_{OUT.S} \# Q_{OUT.R}$$

JK to DE:

The original JK definition provides the $Q_{OUT.J}$ and $Q_{OUT.K}$ equations. These equations are then converted as:

$$Q_{OUT.D} = Q_{OUT.J} \# (!Q_{OUT} \& Q_{OUT.J} \& Q_{OUT.K})$$

$$Q_{OUT.E} = Q_{OUT.J} \# Q_{OUT.K}$$

The term (!Q_OUT & Q_OUT.J & Q_OUT.K) is needed only if the toggle function of the JK is used.

T to DE or D:

The original T definition provides the Q_OUT.T equation. This equation is then converted to DE as:

$$\begin{aligned}Q_OUT.D &= !Q_OUT \\ Q_OUT.E &= Q_OUT.T\end{aligned}$$

The toggle flip flop can also be converted to a D register as:

$$Q_OUT.d = !Q_OUT \& Q_OUT.T$$

One of the main factors for the conversion is that the DE has the same product term and speed efficiency as the original registers. The 6002 macrocell is also selectable with a synchronous or asynchronous clock on an output by output basis. The output logic macrocells outputs either feedback into the AND array through an I/O macrocell or exit the device through an I/O pin.

If the devices output macrocell feeds back into the array the user may still use the I/O pin as an input unlike other PLDs where when a signal is feed back, the I/O pin is lost. The 6002 has 15nsec Tpd and a 6.5nsec Tco. It should be noted that one GAL6002 is capable of directly replacing up to 2 1/2 22V10 with a signal device.

Summary

If control logic function are to implemented in programmable logic devices it is then incumbent on the design engineer to use devices/architectures that best meets their design requirements. The state machine designer should establish a set a criteria by which to choose a solution to a particular state machine challenge. Parameters such as input and I/O count, pin densities, number and type of registers, performance, price and power should all be evaluated when choosing a state machine device.

Avoid the Pitfalls of High-Speed Logic Design

This article is reprinted from *Electronic Design*—November 9, 1989.

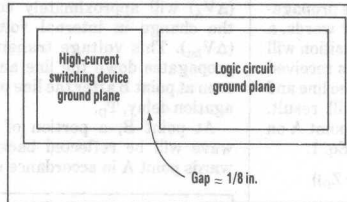
MAKE SURE THAT YOUR TURBO-CHARGED LOGIC SYSTEM WORKS BY PAYING AS MUCH ATTENTION TO PRINTED-CIRCUIT BOARD LAYOUT TECHNIQUES AS TO LOGIC DESIGN CONSIDERATIONS.

AVOID THE PITFALLS OF HIGH-SPEED LOGIC DESIGN

Modern high-speed systems demand modern high-speed logic families. Consequently, semiconductor houses have developed such product lines as ACT, FACT, and AS. But these systems also demand that the lay-out of their boards conform with the results of distributed-element theory, otherwise ringing, crosstalk, and other transmission-line phenomena render those systems inoperative. Meeting this second requirement necessitates something more than a new product introduction—it insists on a change in the way logic boards are engineered. The logic-systems designer and the board-layout designer must work hand-in-hand if a viable high-speed board or system is to be produced.

In the past, logic design and board layout were usually regarded as separate parts of the design process. First the system designer configured the logic, then the board engineer laid it out. That approach worked because slew rates were so low (0.3 to 0.5 V/ns) that crosstalk wasn't much of a problem; rise times were so long (4 to 6 ns) that ringing could settle down before a logic element could change state; and in general, the assumptions of lumped-element circuit theory usually worked out pretty well.

For systems designed with today's high-speed logic circuitry, those underlying assumptions no longer hold true. Today's slew rates are on the order of 2 to 3 V/ns, rise times are below 2 ns (frequently, below 1 ns), and transmission-line phenomena, such as ringing, can be a problem for trace



1. TO MINIMIZE NOISE, THE ground plane should be fragmented into separate areas for noisy high-current devices and for sensitive logic circuits. For best results, the number of signal lines that cross the gap between the fragments should be minimized.

JOCK TOMLINSON

Lattice Semiconductor Corp., P.O. Box 2500, Portland, OR 97208; (503) 681-0118.

Reprinted with permission from ELECTRONIC DESIGN - November 9, 1989

DESIGN APPLICATIONS DESIGNING WITH HIGH-SPEED LOGIC

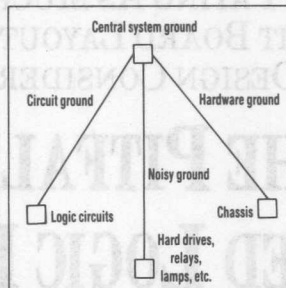
lengths as short as 7 in. As a result, logic designers must take certain steps:

- Use ground and power planes.
- Control conductor spacings to eliminate crosstalk.
- Make extensive use of decoupling capacitors.
- Pay attention to ac loading.
- Terminate lines properly to minimize reflections.

PLANE ADVICE

For high-speed logic, ground planes aren't simply suggested for reliable board performance—they are absolutely necessary. It's essential that one layer of the board be assigned for a ground plane and that it cover as large an area as possible. A solid ground plane lowers the ground-return-path impedance as well as the device-to-device ground pin impedance.

But a common ground plane for all of the circuitry in a system can cause problems by coupling noise from high-current switching devices into sensitive logic inputs. Therefore, the ground plane for such high-current



2. SEPARATE DEDICATED

grounds should be supplied for the logic circuitry, noisy high-current devices, and the chassis. The three should come together at one point, the central system ground, which is usually located near the power supply.

devices as relays, lamps, motors, and hard drives should be separated from the logic ground. This can be accomplished by fragmenting the ground plane into discrete areas (Fig. 1).

But fragmentation causes problems of its own—it creates discontinuities in the characteristic imped-

ance of any transmission line that crosses the separation between fragments. Therefore, for best results, boards should be laid out so that only two fragments are needed. The gap between those fragments should be kept as narrow as possible (an eighth of an inch works well in most applications), and the number of signal lines that cross the gap should be minimized. Designers should also bear in mind that through-holes and vias subtract from the effective area of the plane, increasing its effective impedance.

As with grounding, an entire layer of the board should be designated as a power plane. Even though it is at a different potential, the power plane should be implemented in accordance with the same concepts as the ground plane. Therefore, it should be fragmented when necessary to isolate noisy components from delicate logic circuits.

A WELL-GROUNDED SYSTEM

In addition to properly designed power and ground planes, high-speed logic systems require the establishment of a good, clean (low-

SIGNAL LINES BECOME TRANSMISSION LINES

For the transmission line model illustrated in the diagram, the rise time (t_R) is less than the line propagation delay (T_D). In other words, a complete TTL level transition will occur before the pulse is received at the receiving end of the line and reflections (ringing) will result. The voltage change at point A on the line is expressed in Eq. 1:

$$\Delta V_A = \Delta V_{int} (Z_0 / (R_0 + Z_0))$$

Where: V_{int} = internal voltage on the output of the driver;

R_0 = output impedance of the driving gate;

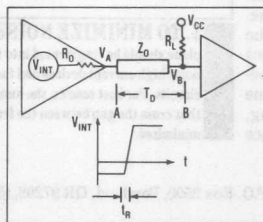
R_L = load impedance;

Z_0 = the characteristic line impedance;

and V_A = the source voltage at the sending end of the line.

Because R_0 is so small when compared to the line impedance, the change in voltage at point A (ΔV_A) will approximately equal the change in internal voltage (ΔV_{int}). This voltage transition propagates down the line and is seen at point B after the line propagation delay, T_D .

At point B, a portion of the wave will be reflected back towards point A in accordance with



the formula (Eq. 2):

Eq. 2

$$\rho_L = (R_L - Z_0) / (R_L + Z_0)$$

where ρ_L , called the voltage reflection coefficient (rho), is the ratio of the reflected voltage to the incident voltage.

After examining Eq. 2, it should be evident that $-1 \leq \rho \leq +1$. It should also be evident that there will be no reflected wave if $R_L = Z_0$ —if the line is terminated in its characteristic impedance. Note that the reflected wave can, in principle, be as large as the incident voltage and of either positive or negative polarity.

This analysis holds true for the sending end of the line, as well as the receiving end. That is,

Eq. 3

$$\rho_S = (R_0 - Z_0) / (R_0 + Z_0)$$

DESIGN APPLICATIONS

DESIGNING WITH HIGH-SPEED LOGIC

noise) system ground for reliable performance. A clean system ground ensures less noise within the system, and thus ensures good, strong transistor margins. At least 10% of the ground connections on the pc card should be connected to the system ground to reduce card-to-ground impedance.

Like the ground and power planes of the individual boards, the overall grounding scheme should be fragmented with separate conductors provided for the various sections of the system. For example, all relays, lamps, hard drives, and other noise-generating devices should have their own separate ground path. The system's mechanical package (chassis, panels, and cabinet doors) should have a dedicated ground. And, of course, the logic circuitry should have a ground of its own.

Those three grounds should then come together at the central system ground point, which will usually be located near the power supply (Fig. 2). This common-point grounding technique can also be very effective in reducing radiated interference (EMI and RFI).

TAMING CROSSTALK

Crosstalk—the undesirable coupling of a signal on one conductor to one on a nearby conductor—becomes an increasingly serious problem as slew rates go up. This signal coupling is made worse if the second trace has a high impedance or if the traces run parallel to one another for more than a few inches and are spaced less than 100 to 150 mils apart.

Crosstalk can be catastrophic to a logic board, sabotaging a conceptually flawless piece of logic design. For example, if a clock line and a data line run parallel to each other for more than several inches, and if the

data line cross-couples or superimposes its signal onto the clock line, the device that the clock is driving may detect an illegal level transition.

Methods to reduce crosstalk are straightforward, though not particularly elegant. The coupling can be attenuated by separating the adjacent traces as much as possible. The trouble with this approach is that available board real estate often lim-

creating a stub or a high-frequency antenna.

Another step that can be taken to reduce crosstalk is to lower the impedance of those traces into which crosstalk is especially to be avoided. The lower the impedance that a trace presents, the harder it will be to cross-couple a signal into it.

Even with the use of power and ground planes on a pc board, decou-

pling capacitors must be used on the V_{CC} pins of every high-speed device. Those devices demand a nearly instantaneous change in current whenever they switch states. Because the power plane can't meet that demand, a high-quality decoupling capacitor is required, otherwise the switching will cause noise on the V_{CC} plane.

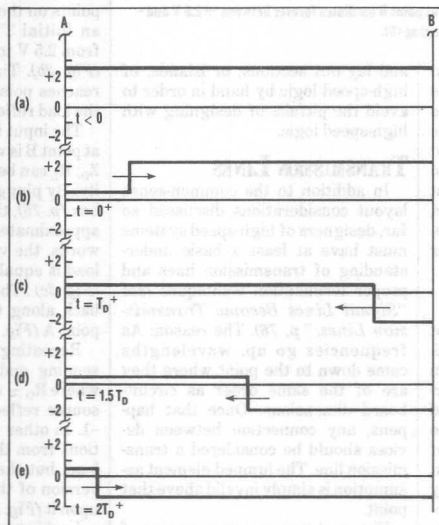
A 0.1- μ F multilayer ceramic (MLC) or other RF quality (low-inductance) capacitor should be placed on every fast-slew-rate device as close to the V_{CC} pin as possible. The commercially available DIP sockets with built-in decoupling capacitors also work well in this application.

Most designers, when they think of loading at all, think in terms of dc loading—traditionally referred to as fan-out and fan-in. But that type of loading rarely presents a problem with today's state-of-the-art logic devices. Much more significant

when designing with high-speed logic are input and output ac loading.

INPUT CAPACITANCE

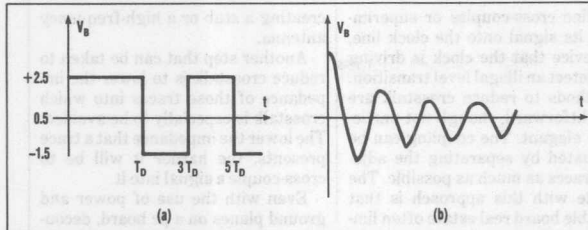
Because the input capacitance of a device impacts the overall performance of the logic circuit, it should be examined before a particular device is selected for a design. To ensure specified performance, the total load capacitance that a device drives—including the distributed ca-



3. WAVE PROPAGATION along a transmission line occurs as follows: Prior to time zero, there is a steady-state voltage of 2.5 V dc on the line (a). At $t = 0$, the voltage at point A drops to 0.5 V, sending a negative pulse of -2 V toward point B (b). At $t = T_D$, that negative pulse is reflected from point B. It adds algebraically to the 0.5 V on the line and sends a -1.5-V pulse back toward point A (c). The reflections then continue as in (d) and (e).

its the possible separation to an inadequate amount.

Ground striping, or shielding, is an effective way to reduce crosstalk and it makes better use of available board area. With ground striping, a ground trace (the stripe) is run between the two parallel traces to act as a shield. If ground striping is used, through holes to the ground plane should be placed every 1 to 1.5 inches along the ground strip to eliminate the possibility of inadvertently



4. IDEALLY, THE VOLTAGE at point B oscillates forever between +2.5 V and -1.5 V (a). In reality, it will be a damped ringing (b).

capacitance of the trace—shouldn't exceed the device's specified capacitive load. Most high-speed logic devices have a maximum loading of 50 pF. As a rule of thumb, the maximum load on any logic element should be no more than four to six devices for best speed/load performance. However, there are some high-slew-rate devices on the market that have higher output drive capabilities.

BEWARE OF AUTOROUTER

The most common reason for not following the board-layout principles mentioned so far is having an autorouter do the layout. Autorouters do what they were designed to do very well: They place traces so as to make the most efficient use of the pc-board real estate. But most autorouters don't have the capability to determine which devices are high-speed and which are not. This is where the logic designer must step in

and lay out sections, or islands, of high-speed logic by hand in order to avoid the pitfalls of designing with high-speed logic.

TRANSMISSION LINES

In addition to the common-sense layout considerations discussed so far, designers of high-speed systems must have at least a basic understanding of transmission lines and proper termination techniques (see "Signal Lines Become Transmission Lines," p. 76). The reason: As frequencies go up, wavelengths come down to the point where they are of the same order as circuit-board dimensions. Once that happens, any connection between devices should be considered a transmission line. The lumped-element assumption is simply invalid above that point.

The most common consequence of failing to consider the distributed na-

ture of a high-speed logic board is ringing, which is caused by multiple reflections from the ends of unterminated transmission lines. An unterminated line has no load impedance ($R_L = \infty$) and is therefore an impedance-mismatched line. The behavior of this line when connected to a device with a fast slew rate can be understood from the following example: Prior to time zero, there's a steady-state voltage of 2.5 V dc at all points on the line (Fig. 3a). At $t = 0$, an initial TTL voltage transition from 2.5 V to 0.5 V occurs at point A (Fig. 3b). Time T_D later, the signal reaches point B and is reflected by the load reflection coefficient, ρ_L .

The input impedance of the device at point B is very high with respect to Z_0 ; R_L can be approximated by infinity. By plugging into Eq. 2 from the box (p. 76), the reflection coefficient approximately equals +1. In other words, the voltage reflected by the load is equal to the incident voltage (Fig. 3c). The reflected wave passes back along the signal path toward point A (Fig. 3d).

Repeating the calculations for the sending end of the line (point A), where $R_0 \approx 0$, you get a value for the source reflection coefficient, ρ_S , of -1. In other words, there are reflections from the source as well as the load, but the source reflects the inversion of the wave that is incident upon it (Fig. 3e).

Looking just at the behavior of the signal at point B, the single-step volt-

RULES TO REMEMBER

The following ten rules summarize everything the logic designer needs to know when designing with high-speed CMOS.

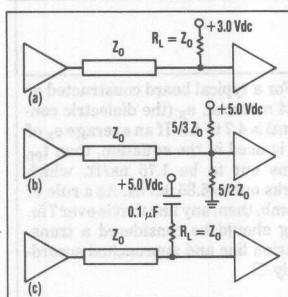
- 1) Keep signal interconnections as short as possible.
- 2) Use a multilayer PCB.
- 3) Provide ground and power planes. Discontinuities in the planes should be avoided because reflections can occur from abrupt changes in the characteristic impedance.

- 4) Fragment the ground and power planes to supply separate sections for high-current switching devices.
- 5) Use decoupling capacitors on every high-speed logic device (0.1 μ F MLC type) located as close to the V_{CC} pin as possible.
- 6) Provide the maximum possible spacing among all high-speed parallel signal leads.
- 7) Terminate high-speed signal lines where $t_R < 2T_D$.

- 8) Beware of ac loading conditions within the design. Exceeding the manufacturer's recommended operating conditions, especially for capacitance, can cause problems.
- 9) When using parallel termination, put bends in all high-speed signal runs that go to more than one load. Use a termination load at the absolute end of the line.
- 10) Create islands of high-speed devices on the pc board. This simplifies board layout and ropes-off the high-speed areas.

DESIGN APPLICATIONS

DESIGNING WITH HIGH-SPEED LOGIC



5. THE BASIC PARALLEL

termination scheme works well but requires a separate 3-V supply (a). The Thevenin equivalent eliminates the need for a separate supply, but dissipates extra power from the regular 5-V supply (b). The use of a capacitor cuts dc dissipation altogether while supplying ac termination (c).

age transition at $t = 0$ leads to an endlessly oscillating signal with a total voltage swing of 4.0 V—twice the original level transition. The voltage doubling comes about because the voltage at point B is the sum of the incident and reflected waves at that point (Fig. 4a). Actually, because of the non-ideal nature of a real circuit board (finite input and output impedances, losses in the transmission lines, and so forth), ρ_L will be less than +1, and ρ_S will be greater than -1. As a result, the reflections will become successively smaller, causing the familiar damped ringing condition (Fig. 4b).

If the ringing amplitude is large enough, it can cause the receiving device to see an illegal level transition and possibly result in spurious logic states occupying the logic design. In some cases, the amplitude of the ringing can actually be large enough to damage the input of the receiving device.

TERMINATE YOUR TROUBLES

The way to eliminate ringing on a transmission line is to terminate the line in its characteristic impedance at either the sending or receiving end. The most common way to terminate a line is with a parallel termination at

the receiving end (Fig. 5).

In the configuration (Fig. 5a), $R_L = Z_0$ and R_L is pulled up to 3 V dc. In principle, R_L could be tied to ground, but TTL-compatible devices could not then supply the necessary drive.

Solving for ρ_L (Eq. 2), it can be seen that $\rho_L = 0$. Terminating a line in its characteristic impedance results in a reflection coefficient of zero, which means that there will be no reflections or distortions on the line. Other than the time delay, T_D , the line will act as if it were a dc circuit. It's important to note that even though devices or gates may be placed at any location on the line, the terminating resistor should be placed at the end of the line. In no case should the line be split like a Tee to feed several devices in parallel (Fig. 6a). Instead, it should be serpentine to feed them sequentially (Fig. 6b).

The 3-V power source shown (Fig. 5a) appears at first to be a major drawback, but R_L and the power supply can be expressed as a Thevenin equivalent running off the system power supply of 5 V dc (Fig. 5b). This variant works well, but the designer should bear in mind that it dissipates additional power.

REDUCING DISSIPATION

A solution that dissipates less power than either of the others uses a capacitor to cut the dc dissipation to zero (Fig. 5c). The recommended capacitor is a 0.1-μF MLC type. Several manufacturers produce both capacitor-resistor and pull-up/pull-down termination packs. The pull-up/pull-down packs usually come in a single in-line package (SIP) with pins on 0.1-in. centers, while the capacitor-resistor combination comes in a standard 16-pin DIP. The most common SIP pull-up/pull-down resistor values are 220Ω/330Ω, 330Ω/470Ω combinations.

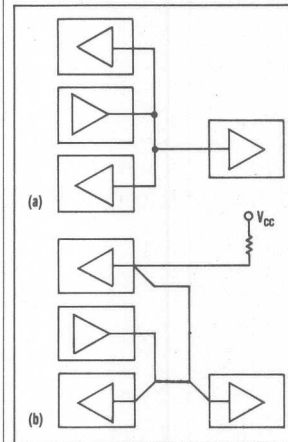
An alternative to a parallel termination at the receiving end is a series termination at the sending end (Fig. 7). The idea behind serial termination is to make $\rho_S = 0$ and $\rho_L = +1$. To do so, R_L is made equal to infinity (left unterminated) and a series resistor is added at the source to make the overall source impedance equal to the

characteristic impedance of the line—that is, $R_S + R_0 = Z_{OL}$.

Making $R_S + R_0$ equal to Z_{OL} , of course, creates a voltage divider, which puts half of the signal amplitude across the line and half across the series combination of R_S and R_0 . Therefore, with the series termination, the amplitude of the transmitted wave is half of what it would be without the termination.

Interestingly enough, the unterminated receiving end of the line precisely compensates for this halving of the amplitude. The reason is as follows: At the receiving end, the half-amplitude wave is received and a half-amplitude wave is reflected. But bear in mind that those are two separate waves whose amplitudes add at the point of reflection. As a result of this addition, the only thing seen at the receiving end of the line is a full-size pulse.

The main disadvantage of a series termination is that the receiving gate or gates must be at the end of the line—no distributed loading is possible. The obvious advantage of a series termination over a parallel one is that a series termination doesn't

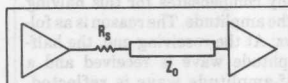


6. SERPENTINING IS essential when terminating a line. Never split the line to feed parallel devices (a). Rather, feed them sequentially with a serpentine line (b).

DESIGNING WITH HIGH-SPEED LOGIC

require any connection to a power supply.

Transmission-line effects must be taken into consideration whenever line propagation delays get up to the point where a signal transition can be completed before that signal can travel down a line, be reflected, and travel back to its starting point. In



7. THE SERIES termination needs no pull-up supply. Its main disadvantage is that it can't handle distributed loads.

other words, lines must be terminated when,

$$2T_D = T_R$$

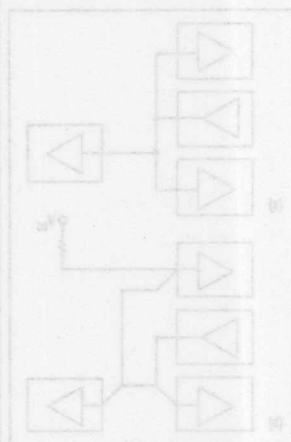
CALCULATING DELAY

Taking 2 ns as a typical rise time for a state-of-the-art high-speed logic device, how long can a board trace get before its propagation delay gets to be 1-ns long? For a pc board with a continuous ground plane and a signal trace on the adjacent layer, the propagation delay depends on only one variable, the dielectric constant of the board material. That delay time is given by:

$$t_{PD} = 1.017 (0.475 e_R + 0.67)^{1/2} \text{ ns/ft}$$

For a typical board constructed of FR4 material, e_R (the dielectric constant) is 4.7 to 4.9. If an average e_R of 4.8 is used in the equation, then t_{PD} turns out to be 1.75 ns/ft, which works out to be 6.86 in./ns. As a rule of thumb, then, any line that is over 7 in. long should be considered a transmission line and approached accordingly. □

Jack Tomlinson, senior applications engineer at Lattice, holds a BSEE from Colorado State University.



8. REPRESENTING IS essential when terminating a bus. The signal line is terminated at both ends with resistors. The devices are connected to the line in a bus configuration.

REDUCING DISAPPEARANCE

A reduction in the disappearance of power from other of the other uses a capacitor to cut the disappearance of power. The capacitor is connected to the signal line and the power supply. The capacitor acts as a low-pass filter, reducing the high-frequency components of the signal. This reduces the power loss due to the disappearance of power.

As an alternative to a parallel termination at the receiving end is a series termination at the sending end. The series termination is a resistor connected in series with the signal line. This resistor is chosen such that the total resistance of the line and the load is equal to the characteristic impedance of the line. This ensures that the signal is properly terminated and no reflections occur.

If the rising amplitude is large enough, it can cause the receiving device to see an illegal level transition and possibly result in spurious logic states occupying the logic design. In some cases the amplitude of the rising edge can actually be large enough to change the input of the receiving device.

TERMINATE YOUR TROUBLE

The way to eliminate trouble on a transmission line is to terminate the line at both ends with resistors. This ensures that the signal is properly terminated and no reflections occur. The most common way to terminate a line is with a parallel termination at both ends.

PLD - Design Methods Migrate Existing Designs to High-Capacity Devices

This article is reprinted from *EDN* - February 17, 1994.

Vendor's EDIF netlist reader may not fully understand vendor's EDIF netlist. Consequently, using incompatible netlists would make your translation at least incomplete and at worst disastrous.

However, translation becomes simple and reliable if you use PLD design methods to express your design. In particular, most designers use Boolean equations to describe their PLD functions because of the AND/OR architecture of PLDs. For example, because PLDs have become extremely popular as a "can-do-everything" circuit element, CAE vendors have integrated PLD development tools into their CAE tools. These tools translate Boolean equations in various formats which you can easily translate from one to another, giving an accurate and complete functional description of a circuit.

To illustrate, consider the high-level description of a 10-bit up/down, presetable counter in Listing 1 (see EDN 1100). This functional description uses ABEL 4.XL, a device-independent language. A device's description at this level does not translate easily to any specific PLD. However, you can use a PLD compiler to transform the

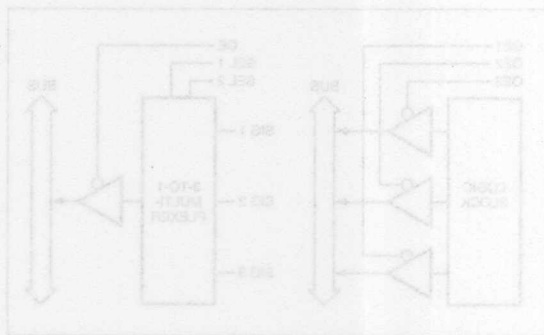


Fig 1—You can translate 2-state logic into 1-of-N selection logic that is implemented in a high-density programmable device.

Moving to newer higher capacity programmable devices can give you higher density and better performance. But if you use common CAE design tools, transferring your old design's description to the new device's format may be difficult. PLD design methods help ensure that your design remains translatable.

All CAE tools accept PLD design methods. Thus, using PLD design methods leads to a very simple method for migrating designs to new types of PLDs. Using these methods helps you take advantage of the increased performance and lower cost of newer devices and lets you combine new designs with existing ones.

Overcoming proprietary databases. Each CAE vendor has developed a proprietary database for storing circuit-design information among its own tool set. To transfer circuit-design information from one vendor's tool set to another's, you must perform a complex translation. This translation may misinterpret or drop information altogether, yielding an erroneous result.

The electronic "industry standard" Electronic Design Interchange Format (EDIF) netlist solves this problem. EDIF can represent circuitry at levels ranging from a complete graphical, functional, and parametric representation to only a primitive functional representation. As a result, one

PLD-design methods migrate existing designs to high-capacity devices

Mike Trapp, Lattice Semiconductor Corp

Moving to newer higher capacity programmable devices can give you higher density and better performance. But if you use common CAE design tools, transferring your old design's description to the new device's development environment may be difficult. PLD-design methods help ensure that your design remains transportable.

All CAE tools accept PLD-design methods. Thus, using PLD-design methods leads to a very simple method for moving designs to new types of PLDs. Using these methods helps you take advantage of the increased performance and lower cost of newer devices and lets you combine new designs with existing ones.

Overcoming proprietary databases

Each CAE vendor has developed a proprietary database for sharing circuit-design information among its own tool set. To transfer circuit-design information from one vendor's tool set to another's, you must perform a complex translation. This translation may misinterpret or drop information altogether, yielding an erroneous result.

The ostensible "industry-standard" Electronic Design Interchange Format (EDIF) netlist typifies this problem. EDIF can represent circuits at levels ranging from a complete graphical, functional, and parametric representation to only a primitive functional representation. As a result, one

vendor's EDIF netlist reader may not totally understand another's EDIF netlist. Consequently, using incompatible netlists would make your translation at least incomplete and at worst inaccurate.

However, translation becomes simple and reliable if you use PLD-design methods to express your design. In particular, most designers use Boolean equations to describe their PLDs' functions because of the AND/OR architecture of PLDs. Fortunately, because PLDs have become extremely popular as a "can-do-anything" circuit element, CAE vendors have integrated PLD-development tools into their CAE tools. These tools produce Boolean equations in various formats, which you can easily translate from one to another, giving an accurate and complete functional description of a circuit.

To illustrate, consider the high-level description of a 10-bit, up/down, preloadable counter in **Listing 1** (see *EDN* BBS). This functional description uses ABEL 4.XX, a device-independent language. A device's description at this level does not translate easily to any specific PLD. However, you can use a PLD compiler to transform the

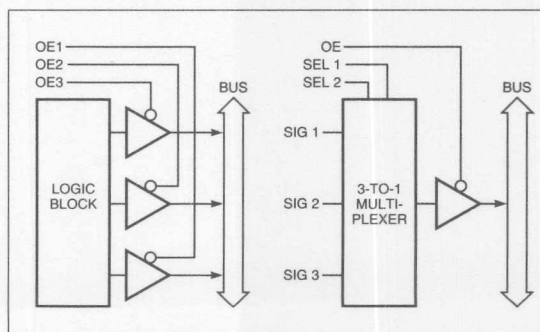


Fig 1—You can translate 3-state buses into 1-of-N selection functions for implementing in a high-density programmable device.

PLD-DESIGN METHODS

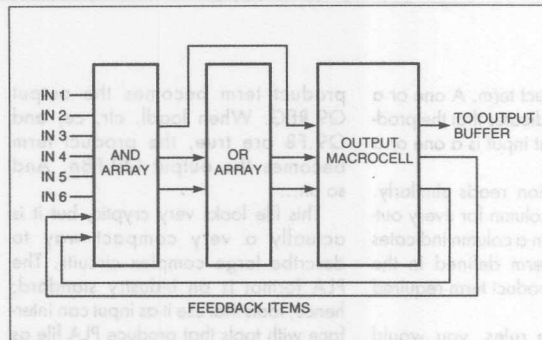


Fig 2—Boolean expansion may improve device utilization but can introduce feedback terms.

high-level ABEL description into reduced Boolean equations (**Listing 2** on *EDN* BBS)—the fundamental description of a 10-bit up/down counter. Now, you can move this Boolean representation of the counter to virtually any development environment with only minor changes.

Because all mainstream PLD compilers produce reduced

equations with virtually identical syntax, migrating these equations from tool to tool is simple. For instance, the PDS tool from Lattice for developing devices in the company's pLSI family has an equation syntax virtually identical to ABEL's reduced-equations syntax. Similarly, the PLA files that ABEL, Minc, and CUPL produce all follow the standard rules for PLA-file syntax.

You can even extend this technique of obtaining reduced, transferable equations from a schematic diagram. Many CAE companies include PLD compilers, with associated PLD libraries, in their design environments. With components from their PLD library, you can use the CAE tools to produce reduced Boolean equations that describe the schematic. You can then integrate these equations with functions expressed in a PLD-description language to form a set of equations that completely describes your circuit. The design tools may produce these equations in the familiar textual form or in an industry-standard format known as PLA (see **box**, "Interpreting PLA files").

If you capture your old designs in the form of these equations, you just follow a series of basic steps to implement them in a new programmable device:

- Define the I/Os.

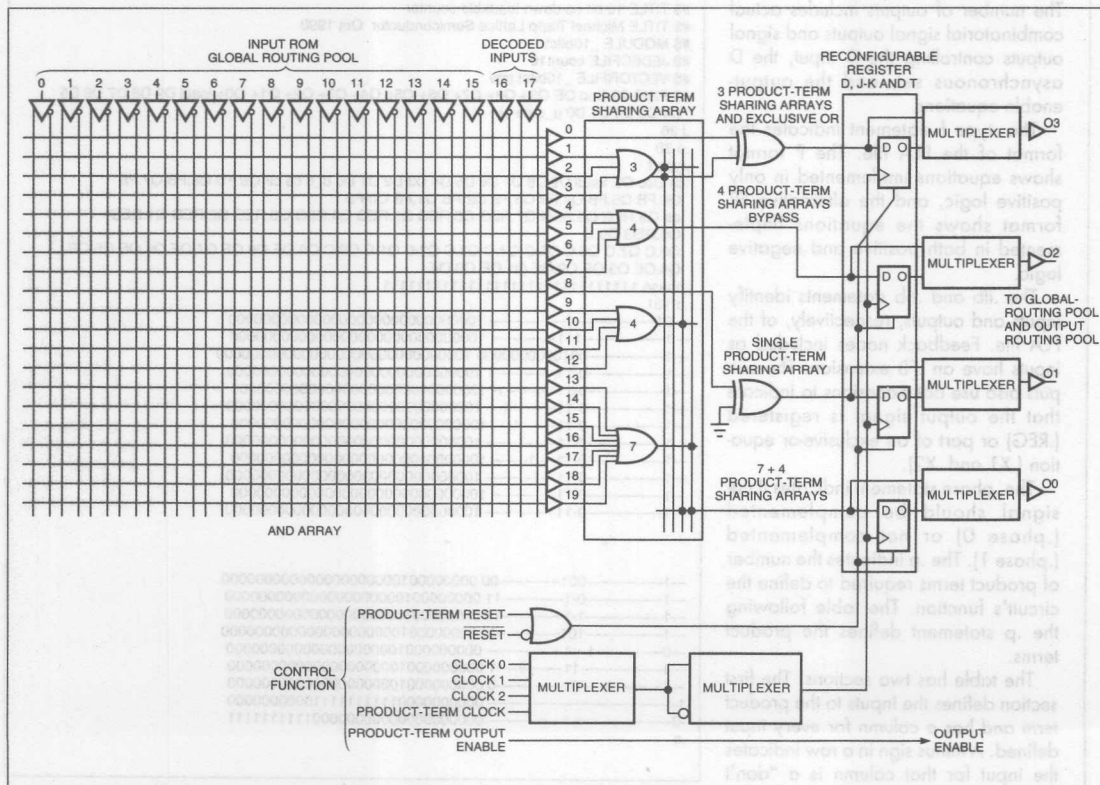


Fig 3—The pLSI logic block allows different product-term-sharing combinations for implementing various logic functions.

Interpreting PLA files

The *i* and *o* statements indicate the number of inputs and outputs, respectively. The number of inputs includes nodes feeding back as circuit inputs. The number of outputs includes actual combinatorial signal outputs and signal outputs controlling the D input, the D asynchronous set, and the output-enable equations.

The .ilb and .ob statements identify inputs and outputs, respectively, of the PLA file. Feedback nodes included as inputs have an .FB extension. The outputs also use dot extensions to indicate that the output signal is registered (.REG) or part of an exclusive-or equation (.X1 and .X2).

The table has two sections. The first section defines the inputs to the product term and has a column for every input defined. A minus sign in a row indicates the input for that column is a "don't

The output section reads similarly. This section has a column for every output defined. A one in a column indicates that the product term defined in the input section is a product term required for that output.

This file looks very cryptic, but it is actually a very compact way to describe large complex circuits. The PLA format is an industry standard; hence, tools that use it as input can interface with tools that produce PLA file as output.

```

$# TOOL ABEL 4.10
$# DATE Mon Oct 5 14:41:38 1992
$# TITLE 10 bit up down loadable counter
$# TITLE Michael Trapp Lattice Semiconductor Oct 1990
$# MODULE _10bitctr
$# JEDECFILE count10
$# VECTORFILE _10bitctr.tmv
$# PINS 26 Osc OE Q9+ Q8+ Q7+ Q6+ Q5+ Q4+ Q3+ Q2+ Q1+ Q0+ loadi D9 D8 D7 D6 D5
D4 D3 D2 D1 D0 u_d ce clr
j 26
o 30
.type f
.lib Osc OE loadi D9 D8 D7 D6 D5 D4 D3 D2 D1 D0 u_d ce clr Q9.FB Q8.FB Q7.FB
Q6.FB Q5.FB Q4.FB Q3.FB Q2.FB Q1.FB Q0.FB
ob Q9.REG Q8.REG Q7.REG Q6.REG Q5.REG Q4.REG Q3.REG Q2.REG Q1.REG
Q0.REG Q9.C
Q8.C Q7.C Q6.C Q5.C Q4.C Q3.C Q2.C Q1.C Q0.C Q9.OE Q8.OE Q7.OE Q6.OE Q5.OE
Q4.OE Q3.OE Q2.OE Q1.OE Q0.OE
.phase 1111111111111111111111111111111111
.p 131
-01-----1-----1000000000000000000000000000000000
-1-----111-----1000000000000000000000000000000000
-1-----001000000000010000000000000000000000000000000000000
-1-----0-11-----1100000000000000000000000000000000
-1-----0-11-----1-1000000000000000000000000000000000
-1-----0-11-----1-1000000000000000000000000000000000
-1-----0-11-----1-1000000000000000000000000000000000
-1-----0-11-----1-1000000000000000000000000000000000
-1-----0-11-----1-1000000000000000000000000000000000
-1-----0-11-----1-1000000000000000000000000000000000
-1-----0-11-----1-1000000000000000000000000000000000
-1-----0-11-----1-1000000000000000000000000000000000
-1-----0-111-----1000000000000000000000000000000000
-1-----0-111-----1000000000000000000000000000000000

-1-----001-----0000000001000000000000000000000000
-1-----0-1-----1100000000100000000000000000000000
-1-----1-1-----1000000000100000000000000000000000
-1-----101-----0100000000100000000000000000000000
-1-----1-1-----0000000001000000000000000000000000
-1-----11-----1000000000100000000000000000000000
-1-----01-----0000000001000000000000000000000000
1-----00000000001111111111100000000000000000
-0-----000000000000000000000000000001111111111
e

```


PLD-DESIGN METHODS

- Compile and minimize PLD equations.
- Add MSI and SSI functions.
- Implement 3-state circuits, inversions, and preset/reset.
- Combine all PLD source files into a single file.
- Partition the circuit over the new device's logic blocks.
- Import and verify the design.
- Place and route the design using tools for high-density devices.
- Assign the I/O pins.

An upgraded PLD design begins with defining the I/Os of the new device based on the circuit developed for the old devices. You must first determine if the new design will be pad-limited (the new device does not have enough I/O pins) or gate-limited (the new device does not have enough internal logic).

If the design is pad-limited, you must choose a device with a higher pin count or partition the design among two or more lower-pin-count devices. Using a higher-pin-count device raises the cost of the implementation and typically results in a large amount of unused internal logic. Multiple lower density devices typically incur a lower cost and better utilization of available logic.

A gate-limited design mandates that you select a higher density device. This choice usually leads to unused I/O pins. You can take advantage of the unused pins to introduce additional functions, providing that the design does not become gate-limited again.

A shotgun approach

As a shotgun approach, you can simply draw a box around a circuit, count the I/O and gate requirements, and select a programmable device meeting this gate and I/O count. However this simple-seeming task can be complex, requiring good engineering judgment of how to best use the high-density device.

Another straightforward method to estimate gate count uses SSI, MSI, and PLD equivalents. By adding the number of these circuit blocks required for a circuit, you can determine if the design fits into a high-density device. For instance, the generic logic block of the pLSI family of high-density devices is roughly equivalent to one-half of a 20V8 PAL device. Extending this approximation, roughly one MSI device or two SSI devices can fit into each logic block.

To partition a circuit implemented with MSI, SSI, and PLDs, look for those nodes that are best suited for interconnection within the new device. These nodes typically travel from one device to only one or two other devices. Assign nodes that connect to many devices to the new device's I/O pins, unless you vacuum all the destination devices into the new high-density device.

This approach eases determining whether you should implement a node within the high-density device or allocate it as an I/O pin. Signals that connect to a device not implemented in the high-density device become I/O pins by default. Naturally, nodes going off-board must become I/O pins of the high-density device.

Clocking affects partitioning

Clocking can also impact how you partition a circuit. If the circuit requires more clocks than are available in one of the

new devices, for example, you should partition the circuit over multiple devices, such that circuits with common clocks are in the same device.

After defining the circuitry to be placed into the new device, you begin converting the design into the new device's format. A typical design contains many PLDs and a few MSI and SSI devices. Most PLDs have an associated source-equation file, the source for design equations to be imported into the device-specific software for the new high-density device.

You should reminimize the old equations in the original PLD design before converting them into the high-density device's format. While designing with PLDs, you typically use only Boolean reduction. Advanced reduction algorithms available with standard third-party compilers can further reduce the number of product terms required for a function. These reductions produce a lower gate count and easier implementation into the high-density device.

In some cases, the original design's documentation may not be available. In such a case, you may have only to access a JEDEC fuse map for some PLDs. You must decompile this fuse map into the source equations and reminimize the equations. The decompilers produce raw equations that have generic names for the equation's variables and parameters:

```
Pin23 := Pin01 & Pin02
# Pin03 & Pin04 & Pin05 .....
# Pin09 & Pin10;
Pin23.OE= Pin06;
```

Although functionally correct, these equations are difficult to read. Using the "search-and-replace" function available on word processors, you can recast the generic signal names to match those on your schematic. Signal naming is important because development software and high-density development software connect signals with common names.

Add MSI and SSI functions

At this point, you should add any MSI and SSI functions that you want in your revised design. You can easily integrate MSI and SSI functions into the new device by creating a PLD-design file that emulates the functions and importing that file into the device's design software. Most MSI functions fit neatly into a PLD, especially when a designer uses a PLD such as a 22V10 PAL device whose flexible architecture simplifies I/O and product-term allocating. The same procedure for SSI devices is applicable, but combining AND, OR, and INVERTERS into the MSI- or PLD-design equations should be very simple.

By using the same names on the inputs and outputs of the MSI PLD as on the schematic, the resulting file becomes ready for converting and importing.

An alternative method for implementing these functions is to find the closest equivalent circuit to the desired function within the macro library provided with the high-density-device software. Both of these techniques aim to develop functionally correct equations that best utilize the pLSI's architecture.

You can then add PLD files containing MSI and SSI func-

PLD-DESIGN METHODS

tions to the files containing the converted PLD equations. The goal is to derive functionally correct equations in standard compiler format.

Convert 3-state signals to multiplexers

Trying to implement 3-state buses within an ASIC or high-density devices can create problems, such as causing outputs to go undefined. In fact, many high-density devices can't implement internal 3-state buses at all. As a solution, you can implement 3-state functions with a 1-of-N select function (Ref 1). The inputs to the selector are the signals that would be tied together on the 3-state bus. The select lines of the selector are the individual 3-state enable signals. This technique, commonly used in ASICs, appears in Fig 1.

You would rewrite the 3-state equations for a 1-of-N selector this way:

```
BUSA_SIG1=SIG_1A
BUSA_SIG1.OE=SIG1_OE
BUSB_SIG1=SIG_1B
BUSB_SIG1.OE=SIG1_OE

BUSA_SIG2=SIG_2A
BUSA_SIG2.OE=SIG2_OE
BUSB_SIG2=SIG_2B
BUSB_SIG2.OE=SIG2_OE

BUSA_SIG3=SIG_3A
BUSA_SIG3.OE=SIG3_OE
BUSB_SIG3=SIG_3B
BUSB_SIG3.OE=SIG3_OE

BUSA_OUT= (!SIG1_OE & !SIG2_OE) & SIG_1A
# ( !SIG1_OE & !SIG2_OE) & SIG_2A
# ( !SIG1_OE & SIG2_OE) & SIG_3A;

BUSB_OUT= (!SIG1_OE & !SIG2_OE) & SIG_1B
# ( !SIG1_OE & !SIG2_OE) & SIG_2B
# ( !SIG1_OE & SIG2_OE) & SIG_3B;
```

The six original equations now appear as the two functions of BUSA_OUT and BUSB_OUT. Note that you do not need SIG3_OE.

Assuming that the SIG_1A and SIG_2A expressions use typical PAL-type equations, they would have this AND/OR structure:

```
SIG_1A= SIGA1 & SIGA2 & SIGA3
# SIGA4 & SIGA5 & SIGA6;
SIG_2A= SIGB1 & SIGB2 & SIGB3
# SIGB4 & SIGB5 & SIGB6
```

Then the selector equation becomes

```
BUSA_OUT= (!SIG1_OE & !SIG1_OE) & SIG1 & SIG2 & SIG3
# (!SIG1_OE & !SIG1_OE) & SIG4 & SIG5 & SIG6
# ( !SIG1_OE & !SIG2_OE) & SIGB1 & SIGB2 & SIGB3
# ( !SIG1_OE & !SIG2_OE) & SIGB4 & SIGB5 & SIGB6
# ( !SIG1_OE & SIG2_OE) & SIG_3A;
```

The AND function on the output enables (SIG1_OE, SIG2_OE) does not increase the number of product terms required to implement the various bus-signal functions. This result holds true for product-term-oriented architectures, such as the pLSI devices.

Investigate inversions

Given the wide variety of device architectures, you should investigate active-high vs active-low internal signals to

achieve the highest utilization of the device's resources. The following equation is an example:

```
!OUT= IN1 & IN2 & IN3 # IN4 & IN5 & IN6;
```

If you can't implement this equation with a hardware inverter, you can use Boolean expansion to produce an alternative:

```
OUT=!IN1 # !IN2 # !IN3 & !IN4 # !IN5 # !IN6;
```

This equation requires seven product terms as opposed to two when implemented into a PLD-type device architecture like that in the pLSI devices (Fig 2). The expanded Boolean equation also requires two extra feedback terms to implement the OR-AND function in an AND-OR device architecture.

Define preset/reset mechanism

A frequently neglected, but nonetheless necessary, requirement for digital designs is a reset mechanism. All state-machine designs should have a known power-up state. If you attach a reset line to all your state-machine registers, such a line would unnecessarily use significant routing resources. The reset mechanism should take advantage of the hardware-reset resources available in the new device. You should remove individual reset signals from your design equations and instead use hardware reset.

Many programmable-device architectures provide only reset and no preset mechanism. In these cases, you can complement outputs requiring a preset signal and still use the hardware reset. Alternatively, you can make the preset function synchronous by adding a preset term into the design's equations.

Last, when placing new logic in the high-density device, you should partition that logic into available logic resources. For the pLSI family, you simply write Boolean equations or use the available macros. With the exception of a few keywords, you can enter the equations just as you would using third-party design tools.

Combining source files and partitioning

The *.DOC files produced by third-party compilers come in an industry-standard format. These files contain reduced equations derived from the source file, JEDEC maps, high-level state-machine language, truth tables, and standard equations. You should combine all the individual PLD and MSI *.DOC files into a single source file for partitioning over the high-density device.

Using a pLSI device as an example, you can collect equations into groups of four outputs to partition the PLD equations to fit into the four outputs of the pLSI device's logic blocks (Fig 5). (You must place headers and trailers around the four equations to direct the pLSI design software to partition the equations into a particular logic block.)

The software then maps the equations into the logic block's 18 inputs, 20 product terms, and four registered or combinatorial outputs. Additional logic capacity results from product-term sharing among the four outputs and an optional exclusive-OR gate fed by a product term and an AND-OR term.

PLD-DESIGN METHODS

The final conversion step is defining the I/O cells. The I/O cell for the pLSI device's definition is

```
SYM IOC IOXX 1;
XPIN XSIGNAME PIN# LOCK#;
IB1/OB1 (SIGNAMEIN/SIGNAMEOUT, SIGNAMEIN/SIGNAMEOUT);
END;
```

(Because the pLSI software routes the pLSI devices according to signal name, it automatically connects all I/O cells to the proper internal nodes.)

Import, verify, and lay out the design

Once you have partitioned your design over the new device(s), you must import the device's source file into the design software for verifying, placing, and routing. You then follow the steps outlined for the device's development environment.

This technique for creating a design for a programmable device builds upon PLD-design methods. In summary, the following guidelines can simplify your efforts:

- Decide if the design is I/O- or gate-limited.
- Choose the appropriate new device.
- Use as many of the original Boolean functions from low-density source files as possible.
- Convert 3-state outputs to 1-of-N multiplexer outputs.
- For reset functions, use the global reset for the entire device or asynchronous reset for specific logic resources.

- Remain within the logic resources when partitioning the circuit.

Following this method brings you the significant manufacturing benefits of new generations of programmable devices: smaller boards, simpler test procedures, faster development, and fewer parts in inventory and assembly. **EDN**

Reference

1. Small, Charles H, "Where CMOS rules, multiplexers slave," *EDN*, August 5, 1993, pg 57.

PLD-DESIGN METHODS

• Remain within the logic resources when partitioning the circuit.

Following this method helps you the significant manufacturing benefits of new generations of programmable devices: smaller boards, simpler test procedures, faster development, and lower parts in inventory and assembly.

L. Hsieh, Chapter 8, "Where CMOS rules, multipliers save," EDN, August 3, 1993, pg. 57.

In-System Programmable Logic in High Volume Manufacturing

This paper was presented at the 1993 PLD Design Conference and Exhibit in Santa Clara, California and appeared in the Conference Proceedings, Track 2.

In System Programmable Logic in High Volume Manufacturing

Jock Tomlinson, Field Application Manager

Lattice Semiconductor Corporation

Introduction

With systems and PC boards continuing to decrease in size with increased logic functionality, combined with the high integration levels of today's logic devices, there has never been greater pressure on board level testability. Traditional board test methodologies are no longer adequate for today's highly integrated systems. Several IC manufactures are attempting to address this problem by supplying devices which actually aid the test engineer in their testing of the board.

The challenge for today's design and test engineers is to design in a comprehensive board test methodology while at the same time reduce the cost of test fixturing. As the PC board becomes more and more complex it becomes harder and more expensive to have a bed of nails test to test each portion of the logic on the board.

Help comes from an unlikely source, In-system programmable logic or ISP HDPLDs. In system programmable logic can aid in virtually every stage of the product design and manufacturing cycle, up to and including installation at the customer. However this paper will focus specifically on the high volume manufacturing and testability areas.

In System Programmability (ISP), the ability to program and reprogram logic devices while "in-system". This concept is being pioneered primarily with High-Density PLDs (hereafter referred to collectively as HDPLDs).

ISP is revolutionizing the system designs of the 90's. ISP is an enabling technology that allows designers to define and develop systems with capabilities previously unachievable. With ISP technology, Virtual Hardware, the concept of hardware as flexible and easy to modify as software, becomes a reality. Hardware functions can be programmed and modified real time to expand product features, shorten system design and debug, simplify field upgrades, and perhaps most importantly, enhance product testability.

Technology Overview

The HDPLDs available on the market today can be categorized into four different and distinct CMOS technologies; Anti-fuse, SRAM, EPROM (UVC MOS)

and E²PROM (E²CMOS). Of these four technologies, only three offer reprogrammability.

Of the three reprogrammable CMOS technologies, only SRAM and E²CMOS provide in-system reprogrammability. UVC MOS can only be reprogrammed after the device has been erased by exposure to UV light (up to 20 minutes erasure time). The following manufactures offer in-system reprogrammability: Lattice Xilinx, AT&T and Concurrent.

In-system programmable and reprogrammable devices can be programmed, erased and reprogrammed while soldered directly to the printed circuit board (PCB). The actual implementation of ISP defers slightly between manufactures but the major concepts are the same. In circuit reprogrammable logic devices program and reprogram using a single 5 Vdc supply and either a serial or parallel programming interface for the loading and programming of binary bit patterns (JEDEC files). Conversely standard programmable logic devices require a super voltage (typically over 12 volts) to be applied to program and erase.

Reconfigurability for Test

Testability

Device board level testability is becoming the limiting factor in the high-tech manufacturing arena, the success or failure of a state-of-the-art product often depends upon the time required to build that product. In the case of products incorporating dedicated microprocessors, data transmission circuitry, or other complex electronic hardware, most of the time-to-build is consumed by testing and integration.

Advances in packaging technology have allowed the development of smaller and more dependable carriers, and have facilitated the onset of extremely high density, "lights-out", automated manufacturing. Advances in a number of interrelated areas (such as IR soldering, pick-and-place, adhesives, sensor technology, etc.) have opened the way to high density assembly techniques that would have been considered impossible only a few years ago. Although not commonplace, some size and/or weight critical

products are currently built using a number of unique bulk-reducing construction techniques. Assembly processes which effect double-sided surface mount, chip-and-wire with epoxy cover, dense pack SIPs, or sandwich-mounted flat-packs are all valid means of producing a smaller, lighter, and more reliable final product. Unfortunately, highly advanced high density products are, at best, painfully difficult to test, and can be utterly impractical to repair.

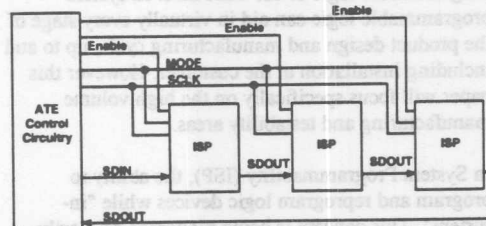
In the manufacturing environment, where replacement parts, known good (golden) prototypes, and sophisticated test equipment are available, the major bottleneck to testing is related to circuit access. Quite simply, many device packages and the boards or sub-assemblies in which these packages are incorporated have few to no internal access paths. This is due, in some cases, to the density and placement of device-to-board interconnects (i.e., adjacent, stacked, double-sided, sandwiched, or epoxied surface mount...). In other cases it can be a result of package pin-count limitations (de-rated LCC, PLCC, pin-grid, or ceramic hybrid packages...). In rare instances, it can even be due to manufacturing or marketing concerns (sync, async, or mixed state machines with secured control patterns and/or "trap" states...). It becomes literally impractical to attempt to drive or receive test signals to or from a unit under test (UUT) from any physical location other than the designed-in system-level contact points.

This situation often renders externally applied test solutions inefficient or unreliable. The consequences, to automated test, are clear. Feedback loops cannot be eliminated. Test (stimulus) vectors cannot be inserted. Response vectors cannot be captured. Digital logic "lumps" cannot be simplified. State machines cannot be reduced. Asynchronous control paths cannot be opened or manipulated. By default, the product or sub-assembly will be verified via some form of BIST (built-in self test) or functional (power-up) diagnostics or may not be tested at all. BIST, even when implemented during the early stages of a design, is expensive in terms of design time, real estate, and parts count. Functional diagnostics (F-diags), at either the system or sub-assembly level, are expensive in terms of development and support. Non-test is expensive in terms of raw yield. None of these solutions are either comprehensive or comfortable. As circuit complexity continues to increase, driven by the synergetic advances in electronic technology mentioned above, the bottleneck(s) associated with high volume manufacture cannot but worsen. Existing approaches to BIST, DFT (Design For Test), and F-diags, will

become more expensive and less effective. Current UUT access techniques will become more costly and less reliable. Only a radical departure from traditional automated test practices and procedures will be able to provide a cost effective solution to tomorrow's test, diagnostic, and verification challenges.

Help comes to the test and manufacturing community from an unlikely source, in system programmable high density logic devices. The ISP approach to BIST was to add serial networking capability to the PCB. With the addition of ISP HDPLDs, independent synchronous serial port capability can help uncover hidden logic. The result is an entirely new approach to testability, RFT (Reconfiguration For Test). Like the JTAG-MSDS 4-wire serial interface (a.k.a. IEEE P1149.1 proposal), the ISP protocol defines a clock input, a serial data input, a mode/control input, and a serial data output.

The serial output of one device can be connected to the serial data input of another, thus allowing an unlimited number of devices to be cascaded.



(NOTE: The ISP serial communication protocol is NOT JTAG-MSDS compatible! ISP devices do not incorporate compliant JTAG-MSDS instruction, data, or identification registers.) Also like the JTAG-MSDS 4-wire serial interface, the ISP protocol can be visualized as an on-board, synchronous, serial FIFO (First-In, First-Out), ring style, local-area-network for devices. An off-board host (or communication master) can communicate with any device on the ring by noting its' relative position and by appropriately transmitting data through the FIFO loop. An ISP device can be completely erased, completely examined, and completely repatterned via the ISP protocol. No unusual voltages or control signals (beyond the signals necessary to exercise the FIFO) are required.

All the voltage/current sources, timing control circuitry, pulse generators, and algorithmic state

generators are built into the device itself (EECMOS). This capability allows ISP based hardware to be modified, or reconfigured, at any point in the life cycle of any given unit or sub-assembly. Thus an extremely complex circuit, composed of interlocked state machines of different types, can be reconfigured into a simpler, more easily tested, configuration. In the factory, feedback loops can be re-routed or eliminated. Latches and/or memory elements can be reduced or isolated. Signal paths into the core of an inaccessible "lump" of hardware can be opened. Test vectors can be introduced without back-drive. In brief, all of the testability problems associated with HDPLD based designs which include feedback or asynchronous paths, can be addressed without the need for physical access to an implementation.

How ISP can solve potential test problems, let us look at the manufacturing needs of a large-volume, medium scale, high speed digital logic design. For the sake of argument, we will assume that all DFT (design for testability) SSI and MSI functions have been implemented using ISP HDPLD devices. This means that all the ATE/ATP (automatic test equipment/automatic test procedure) defeating nuances of a typical small to medium scale logic design can be circumvented. We will further assume that this is a consumer application, and that in-system logic reconfiguration, field service access, and design security are not an issue. This means that the ISP protocol interface(s) can be brought to a spring pin (pogo-pin) compatible pad array on the board or substrate, and that electrical access can be achieved via a typical vacuum or clamp type fixture. To be brief, we will describe the design application as a "board". In reality this application could as well be any type of module or assembly incorporating logic devices. With any PCB or circuit assembly there is a quantifiable level of testability that is achievable, which is dependent on the complexity of the board design.

In an attempt to quantify this uncertainty, a board is typically assigned a "fault-coverage" rating. Good (highly testable) boards are rated at 70% to 99%. Bad (difficult to test) boards are rated at 70% and under. The rating is usually described as "stuck-at fault coverage". This is due to the theory that any function-inhibiting failure can be traced to a "stuck" node or equivalent. Given enough time and effort, anything can be tested well. The reason many boards receive a low fault-coverage rating is because:

(1) The resources necessary to improve the rating by creating a better test exceed the anticipated savings which will be realized throughout the product's life cycle by repairing defective boards identified by the better test.

(2) The resources necessary to perform an improved test on a given lot of boards exceeds the anticipated savings which will be realized by repairing defective boards identified by the improved test.

NOTE: This analysis often does not include the cost to salvage defective boards identified by the customer or by the end-user.

Boards which fail in service must often be replaced or repaired at any cost. Thus, in the case of an unusually long product life cycle, there may be ongoing and increasing pressure to improve the fault-coverage of a given board or sub-assembly due to a high field failure rate. Statistically, the logic circuit configurations which most often cause a loss of confidence in the functionality of a new-board are typified in the implementation of a multiple-device state-machine.

The design function of an ISP device is to permit the device to be repeatedly reprogrammed, via the programming interface, after permanent installation. This provides two important secondary capabilities which, as a side-effect, eliminate the loss of confidence which HDPLDs, programmed and arranged as multiple-device state-machines, normally introduce into a logic design. Essentially, any multiple-device state-machine, or any logic circuit which incorporates DFT violations normally associated with a multiple-device state-machine, can be effectively tested if the components have an ISP compatible interface.

ISP HDPLDs have the capability of being externally interconnected (SDOUT to SDIN). This allows the ISP device to be configured in a serial cascadeable arrangement. The ISP "daisy chain" allows all ISP compatible devices to be verified independent of the function or placement of any given device. The ISP protocol interface is a very simple, low speed, synchronising, which can be quickly and easily verified by ATE or by entry-level test personnel. Given a verified "daisy chain", confidence in the functionality of the individual devices in the chain approaches 90%. Thus the devices which were previously the most difficult and expensive to verify have become the easiest and most cost effective to verify.

Once the chain is verified, any or all of the devices in the chain can be reprogrammed with special self-test patterns, or may be reprogrammed to accept ATE originated stimuli or to drive ATE receivers.

Given a "daisy chain" which has accepted a stimulus and generated a response which does NOT depend upon the design function(s) of any adjacent non-ISP devices, confidence in the functionality of the individual devices in the chain exceeds 90%. Thus the portions of a design which previously exhibited the worst fault-coverage rating now exhibit the best rating.

A verified ISP "daisy chain" greatly facilitates access to and verification of adjacent non-ISP devices and modules. An entire ISP chain or any portion or device may be programmed such that all inputs and outputs remain in a high impedance state whenever the device is powered-up or put into the programming mode. This capability allows a board to be divided into small "lumps" of logic which tremendously reduces the resources required to generate test software. A major cause of new-board mortality, excessive back-drive current, can be completely eliminated via the thoughtful placement of ISP compatible devices by an RFT (reconfiguration for test) conscious design engineer. In the event of an unusually difficult to test "lump", an ISP chain can be programmed to serve as a source of elementary test stimuli. Simple counters, decoders, ATE controlled enable/disable signals, ATE controlled read/write signals, and the like, can be reprogrammed into an ISP chain via the ISP interface.

This new functionality can be entirely dedicated to an intermediate step in the test process for a new-board, and the chain later returned to its primary function after the non-ISP portions of the new-board are satisfactorily verified. This capability, is referred to as: Reconfiguration For Test (RFT)

Unfortunately, such practices extract a high price in terms of production costs (extra gates, solder holes, board area, etceteras) and in terms of performance (5ns to 30ns per ATE controllable gate, infant mortality due to backdrive related stress). In the absence of traditional DFT, the price extracted is in terms of fault coverage and diagnostic engineering resources. These problems can all be solved by using ISP protocol devices where ever feedback signals need to be generated or interpreted. RFT allows feedback loops to be opened, eliminated, or tied to a test node. Given an unused input and an unused output on an ISP protocol device used to generate a feedback signal, that signal can be routed to the unused output where it may be

sensed by ATE without influencing the UUT.

Additionally, the unused input can be routed to the portion of logic which is normally driven by the feedback. Thus the ATE itself is made into a series component in the asynchronous feedback loop. Once the feedback signal has been examined by the ATE, a replica can be created and driven back into the logic normally driven by the feedback signal.

This serves the intent of the traditional DFT requirement for physical interruption of asynchronous feedback loops, but without the need for switches or jumpers. It allows an electrical interruption in the feedback circuit, but without propagation delays due to extra gates, and without potential stress failures due to excessive backdrive. Overall, the use of ISP devices in critical asynchronous feedback circuits allows:

- (1) A reduction in time required to achieve acceptable fault coverage;
- (2) A reduction in resources required to achieve acceptable fault coverage;
- (3) No performance penalties;
- (4) No backdrive overstress;
- (5) Minimal additional hardware;
- (6) Full DFT compliant loop control and interruption.

Reduction of tight hardware kernels In any system design there are invariably certain sections or modules which are uncompromisingly speed critical. An example of such a speed critical logic module or sub-module would be the address decode and access arbitration circuitry for a multiple-port cache RAM bank. Fundamentally, the response time of such a circuit is so critical that no allowance can be made for added functionality or for control which does not enhance the primary design goal or which, at a minimum, incurs no performance overhead. This includes any circuitry which might facilitate testability. As a rule, such circuits constitute less than 20% of a typical system's real estate and consume more than 80% of the diagnostic resources applied to that particular system. This is not a comfortable situation, but until recently the economics which evaluate the return on diagnostic-related expenditures (versus the life-cycle of a system or viability of a manufacturing process) have mandated this style of diagnostic resource allocation. The usual approach to testing such circuitry (since it is known in advance that some compromise in both fault coverage and fault isolation will have to be made), is to attempt to model the group of devices that make up the performance

critical portions of the system as though they were a single MSI or LSI full custom component.

This requires considerable ingenuity on the part of the diagnostic engineer since he/she must generate a complete set of test software/patterns for this pseudo-device as though it were an outside vendor's unsupported new product. This approach can be tremendously improved via the use of ISP HDPLDs to implement the performance critical portions (and therefore the least ATE accessible portions) of any given system. Since ISP HDPLDs are state-of-the-art CMOS programmable logic devices, there is no performance penalty. A "threaded signal" performance critical logic circuit can be completely repatterned, many times, during the performance of an ATE program, to allow the ISP protocol devices to be fully tested. Also, given a high enough percentage of ISP protocol devices in any group of devices which make up a performance critical design, even non-ISP device testability can be improved by using the ISP HDPLDs to implement ATE access paths during the execution of an ATE program.

Although more diagnostic resources (testability guru time) are required to implement an ATE program which includes ISP device RFT control array patterns, the overall savings in time required to test a tight, performance critical hardware module, is actually reduced. This allows increased fault coverage and greatly increased fault isolation, but with an overall reduced demand for resources. Trade-offs can be made which allow greatly increased testability without the expenditure of extra resources, or which provide a net savings in diagnostic resource utilization without any testability loss.

Using ISP to increase visibility into the board in the previous RFT discussions, little mention has been made regarding the tremendous opportunities offered by ISP protocol devices for increased visibility into a new-board during the ATE/ATP (automatic test equipment/automatic test procedure) portions of the manufacturing process. Points mentioned earlier deal mainly with single function, lumped logic modules, constructed partially or even entirely of ISP protocol devices. While this perspective encourages increased fault coverage and better fault isolation of performance critical circuits and/or multiple-device state-machines via the introduction of RFT principles, it does not adequately describe the benefits available by using ISP protocol devices to partition an entire new-board or system for the purpose of enhancing system-level testability.

Most system-level designs incorporate a variety of special-purpose devices (example: RAMs, ROMs, UARTs, controllers, processors, etc.) which are nestled among, and interconnected by, a large quantity of general purpose ("glue" chips) devices. In the last few years, PLDs and HDPLDs have come to replace many of the older SSI and MSI "glue" devices. But for most practical purposes, a HDPLD can be conceptually classified as a multiple SSI and MSI devices breadboard in a package. From a testability viewpoint, HDPLDs offer a reduction in "glue" device package count, but not in logic complexity or in density. In an average large-volume, medium scale, high speed digital logic design, HDPLDs will often account for 20% or more of the logic device package count. HDPLDs, in this type of application, are not famous for improving either the fault coverage or the fault isolation of a logic design. ISP HDPLDs, however, offer exactly this benefit. Since ISP HDPLDs can be serially reprogrammed, it is possible to verify the correct operation of a ISP device's internal logic CA (control array, or fuse map) via programming interface "daisy chain". Verification of any given device's CA implies a very high probability that the entire device is functional. Once an entire chain has been verified, this string of reprogrammable logic devices, extending into a logic design, can be used to improve the visibility into a new-board.

The simplest means of improving visibility is to use the ISP HDPLDs as test signal routing switches. Given a single ATE accessible input to an ISP device, any or all of the device's outputs may be programmed to track that input. Likewise, given a single ATE accessible output, any or all of the device's inputs may be programmed to be tracked. If the ISP HDPLDs have interconnected inputs and outputs, test signals can be routed in and out through any number of discrete ISP devices. (Note: This type of testability enhancement generally requires that a design be implemented with RFT in mind). More complex, is the use of a verified ISP chain to generate simple algorithmic test patterns for the stimulation of non-ISP devices down-stream. Since most HDPLDs perform very well as sequential synchronous state-machines, a device or series of devices with one or more ATE accessible inputs can be programmed to generate a deterministic output pattern in response to any combination of ATE generated clock or data signals. In other words, an ISP HDPLD can be used as part of an ATE/ATP test solution for difficult-to-access non-ISP portions of a logic design. (Note: This type of testability enhancement always requires that a design

be implemented with RFT in mind.) Still more complex, is the use of a verified ISP chain to capture simple synchronous response signals from non-ISP devices up-stream. Because registered outputs work very well as serial shift registers, a device or series of devices with one or more ATE accessible outputs can be programmed to capture several sequential logic values sensed by any or all of the device's inputs. In other words, an ISP HDPLD can be used to capture the results of several successive test patterns and can even perform elementary processing to facilitate signature analysis. Ultimately, a multi-function (generic) system can be designed such that one of the target operations for the design is to allow some form of ATE to thoroughly exercise a thorough subset of all possible logical functions. Such a system would be flexible (due to RFT capabilities) enough to allow every device-external signal conductor to be individually exercised.

The ATE should be able to apply a combination of random and tailored test vectors to any device or multiple-device hardware kernel and to sense the subsequent responses via any signal conductor. This type of testability enhancement would utilize ISP HDPLDs as though the ISP devices themselves made up a serially accessible distributed test processor, and as though the logic CA patterns for a chain of ISP devices acted like individual test processor instructions.

We have seen how ISP HDPLDs can help in the testability of complex logic boards, however there are still several other areas in the product life cycle that also benefit from ISP.

In System Reprogrammability at the Prototype Stage

During any system design cycle, major board building blocks such as microprocessor and RAM are selected first. Decisions regarding system logic tend to be deferred to the later stages of the design process. When using ISP devices, the designer can fully populate his prototype board with its major building blocks, interconnecting all functions with programmable logic. Design changes, whether they require added or modified logic, can be made in minutes using ISP HDPLDs.

Manufacturing Advantages

At present, there are no auto-handlers capable of handling the programming of the higher pin counts associated with today's HDPLDs. As a result, all non-ISP high pin count devices must be programmed by hand, using a standard logic programmer.

It is a non-trivial task to insert a high pin count, small lead pitch device into a programming socket adapter, program, label (or mark) and re-inventory the device without bending the delicate package leads or pins. Auto-inserting devices also increases the risk of exposing the devices to potential ESD environments.

Field Upgrades

ISP HDPLDs provide an ideal way to reconfigure boards and/or upgrade product features in the field. Using conventional logic technology, once a system is installed at a customer location, it becomes very expensive and difficult for the supplier to upgrade the customer to the latest hardware revision, fix hardware bugs or enable hardware options.

HDPLD Device Security

Most ISP HDPLD devices, even though programmed on board, can still assert the security feature eliminating the risk of the JEDEC pattern being read out of the device. If the ISP requires a new or updated JEDEC pattern, the device is erased (which is done automatically before the programming), a new pattern is programmed into the device and the security cell is reasserted.

Conclusion

The challenge for today's design and test engineers is to design in a comprehensive board test methodology while at the same time reduce the cost of test fixturing. As the PC board becomes more and more complex, it becomes harder and more expensive to have a bed of nails test to test each portion of the logic on the board. Therefore the test engineer must now work hand-in-hand to find solutions to these problems. Fortunately, there are options available like ISP HDPLDs to assist in the debug and manufacturing test of complex PCBs.

Higher product quality and reliability can result from the superior test coverage ISP offers. Special test logic can be programmed temporarily into the hardware to facilitate exhaustive product testing. The elimination of defects at an early stage of board check-out reduces more expensive system-level failures later in the final manufacturing process.

ISP HDPLDs are opening doors of opportunity in almost every facet of systems design and test.

A Token Ring Network Adapter Card

Sid Gilbrecht, Senior Design Engineer
3Com Corporation

Introduction

3COM developed a new Token Ring adapter module to make its series of bridge/routers more capable and competitive. To meet tight development schedules, the new module borrowed components and designs from a Token Ring adapter module in another product line. But to meet density and performance requirements, much of the logic had to be integrated into high-density logic devices. To evaluate the devices, the designer developed a model of the critical path circuitry and tried implementing it on a choice of programmable logic devices. Finding a new device that would meet the criteria, the designer developed a series of prototype boards to meet software and QA efforts as well as the shipping date.

Need for Token Ring adapter module

3COM is a leading independent global network company. It provides multivendor connectivity that spans organizations and businesses worldwide. The company designs, manufactures, markets, and supports networking systems based on industry standards and open systems architecture.

3COM's products provide the infrastructure that connects computer systems in a network configuration, enabling businesses to share information within a workgroup, across a campus, or around the world. These connections are possible with end systems—network adapters and terminal servers—which provide the physical links between users and the larger network, and the intermediate systems—internetworking systems and wiring hubs—which connect multiple users or groups within the network. In addition, 3COM provides software, network management, and service and support that bind the products altogether.

A new Intermediate Systems product—called NETBuilder II—was recently introduced

based on 3COM's NETBuilder series of bridges and routers. NETBuilder II is a high-performance, modular-designed bridge/router based on an AMD 29K processor tied to a high-speed proprietary backplane bus. The bus accepts a mix of Local Area Network (LAN) and Wide Area Network (WAN) adapter boards or modules. A NETBuilder II chassis contains either four or eight slots in which users specify adapter boards to form unique configurations for their network needs.

At the time of first shipment, NETBuilder II could only take advantage of two adapter cards—one for Ethernet, and one for FDDI—along with a few WAN boards. For competitive reasons, it was critical for 3COM to increase the LAN coverage for NETBuilder II. Fortunately, 3COM had a Token Ring adapter module for its original NETBuilder series of bridge routers. That design served as a basis for the Token Ring adapter card for NETBuilder II.

Existing Token Ring card

The original NETBuilder Token Ring adapter module was a lower-performance system compared to the specified NETBuilder II design. The module contained two Token Ring channels, as well as two WAN connections, on a board roughly four times the size of the NETBuilder II boards. An on-board microprocessor controlled all four network adapter circuits, in effect creating a four-way bridge.

The NETBuilder adapter module used VLSI chips to implement some of the major functions that would also be found on the NETBuilder II module. First, the Texas Instruments (TI) TMS 380 Token Ring LAN chip set implemented the Token Ring mechanism through its integrated protocol support and system interface engines. The same implementation migrated to the NETBuilder II adapter.

In another example, the NETBuilder adapter board implemented a packet filter to prevent local LAN traffic from reaching the Am29K processor and crossing the bridge. Its design used a TI source route adapter (SRA) chip contained in a single Actel FPGA called the Source Route Transparent Filtering Engine (SRT). This circuit filters source-routed (SR) Token Ring packets: this type of packet contains route information describing the rings and bridges that it must travel to reach its destination.

Much of the rest of the logic was implemented in a bank of 10 low-density PLD devices from Altera and Lattice Semiconductor. This logic included another packet filter for transparent routed (T) packets, which contain only a source destination from which their paths must be discerned.

To operate in the NETBuilder II product, the existing design had to be modified to meet a

variety of new constraints: a new processor interface; a smaller form factor; increasing cost pressures; and a 1K I/O address map limitation on the backplane bus. These modifications had to occur within an extremely short hardware development time—just 3 months. One critical decision that allowed us to meet all these goals was the choice of high-density programmable logic devices (PLDs) to integrate some of the control and datapath logic.

Requirements of new TR card design

Meeting the physical, electrical, and marketing objectives of the NETBuilder II bridge/routers demanded much of the TR adapter module design. For example, the module size as specified in the NETBuilder II backplane was just 8.75 inches in length by 3.9 inches wide. This small size necessitated the use of surface-mount technology on both sides of the

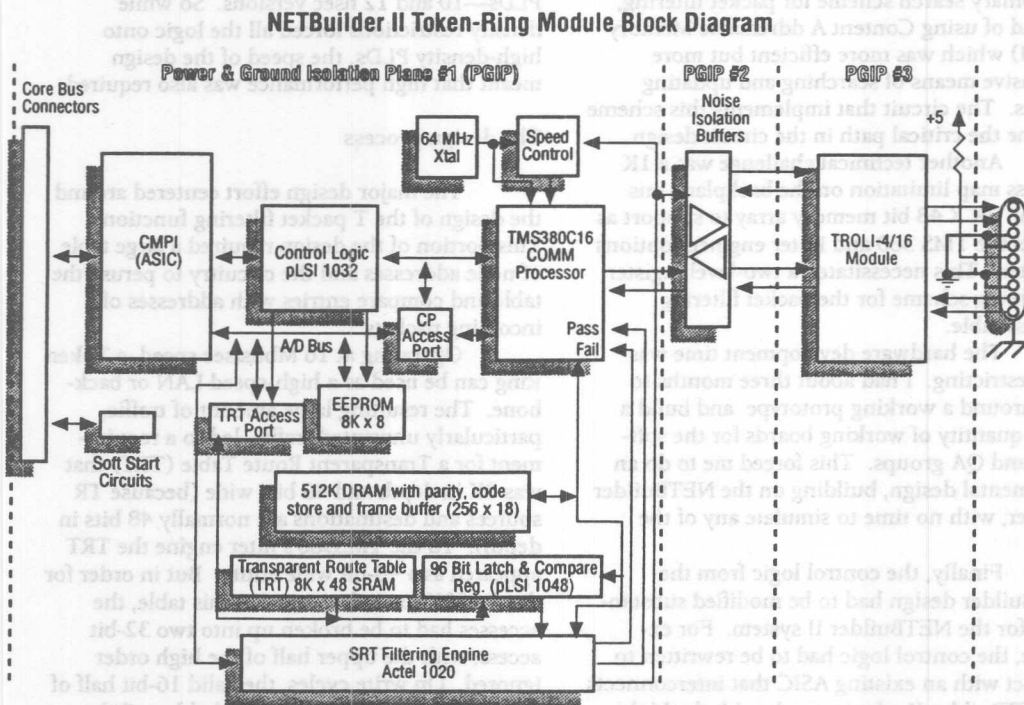


Figure 1: The NETBuilder II Token Ring adapter module design uses the TMS380C16 for the Token Ring mechanism, the CMPI ASIC for interfacing to the high-speed system bus, and an existing SRT filtering engine chip. Other major blocks of logic are contained in two high-density PLDs.

¹ How many?

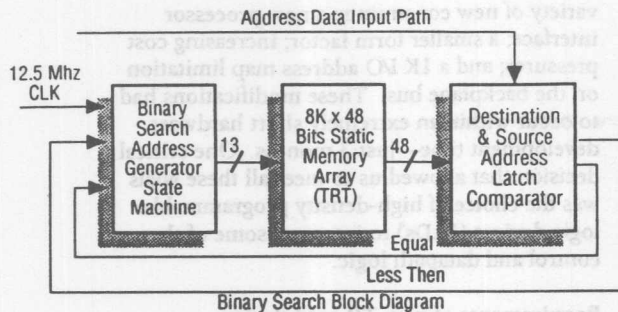


Figure 2: The critical path in the NETBuilder logic is this Binary Search state machine. It must present an index to memory, compare the resulting addresses, and generate a next index based on the results within the 80 nsec cycle time.

board. Even so, high-density PLDs were necessary to incorporate all the control logic.

Despite the high performance requirements, the board had low cost-to-build restrictions. This forced me to go with a much lower cost binary search scheme for packet filtering, instead of using Content Addressable Memory (CAM) which was more efficient but more expensive means of searching and updating entries. The circuit that implements this scheme became the critical path in the circuit design.

Another technical challenge was a 1K address map limitation on the backplane bus with an 8K X 48 bit memory array to support as well as the TMS 380 and Filter engineer options registers. This necessitated a two-level register addressing scheme for the packet filtering address table.

The hardware development time was also restricting. I had about three months to turn around a working prototype and build a small quantity of working boards for the software and QA groups. This forced me to do an incremental design, building on the NETBuilder adapter, with no time to simulate any of the PLDs.

Finally, the control logic from the NETBuilder design had to be modified substantially for the NETBuilder II system. For example, the control logic had to be rewritten to interact with an existing ASIC that interconnects all NETBuilder II adapter cards with the high-speed bus and the Am29K.

Even so, the NETBuilder II adapter module design (Figure 1) was able to use many of the same VLSI components as the NETBuilder

adapter module, such as the TMS380 and the SRT chip to filter out the SR packets. The filter for T packets used the custom NETBuilder design as a basis. These filters preserved backplane bandwidth and Am29K processing time for other adapter modules.

The T Token Ring packet contains only a destination address used to determine its path. The TR adapter module, therefore, had to maintain a table of local LAN addresses so when a T packet was received by one side of the bridge both the destination and source addresses could be looked up in the table: if both were found the packet would be discarded. If either address was not found the packet would be copied to the 29K so that the unknown source address may be learned and placed in the table, or the packet with the unknown destination address could be sent across the bridge.

On the NETBuilder board, the logic was implemented on a bank of relatively high-speed PLDs—10 and 12 nsec versions. So while density restrictions forced all the logic onto high-density PLDs, the speed of the design meant that high performance was also required.

The design process

The major design effort centered around the design of the T packet filtering function. This portion of the design required a large table of node addresses and the circuitry to peruse the table and compare entries with addresses of incoming packets.

Operating at 16 Mbits/sec speed, a Token Ring can be used as a high-speed LAN or backplane. The resulting large amount of traffic, particularly unwanted traffic, led to a requirement for a Transparent Route Table (TRT) that was 8K in depth and 48 bits wide (because TR sources and destinations are normally 48 bits in depth). To the TMS380's filter engine the TRT appeared as a 48-bit wide entity. But in order for the Am29K to read or write to this table, the accesses had to be broken up into two 32-bit access, with the upper half of the high order ignored. On write cycles, the valid 16-bit half of the word had to be latched and held until the low order 32 bits were latched so that the entire 48 bits would be written at once into the array. On a read, all 48 bits were read at once: the high order 16 bits were passed directly to the

NETBuilder II Token-Ring Module Control pLSI® Block Diagram

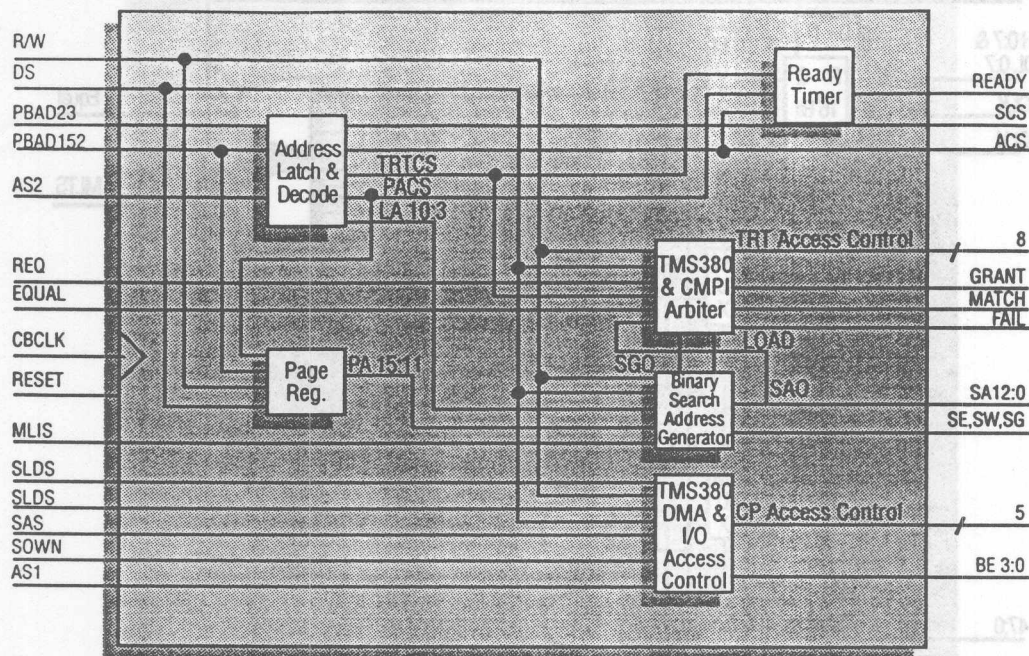


Figure 3: One high-density PLD (pLSI 1032) holds all of the random logic including I/O select, buffer control for both I/O and DMA, the TRT page register, and the binary search state machine address generator.

backplane while the lower 32 bits were latched and held for the next sequential read. Due to the 1K I/O address space, a page register had to be implemented in order for the 29K to access the full 8K of the TRT. The page register had to be 1 byte by 5 bits to allow for 512 words of contiguous TRT space by 32 pages to cover all 8K (16K at 32 bits) of space.

The base system clock ran at 12.5 MHz, an 80 nsec cycle. While this was not a very short cycle it did present a challenge to the perceived critical path: the binary search state machine. It needed to run through a number of logic levels to make a correct decision on the address being compared in the TRT, and generate an appropriate next table address.

Following the diagram in Figure 2, the logic critical path was as follows: First the 29K had initialized the TRT with a sequentially ordered series of local node address, and the TMS380 had latched a TR destination or source

address in the comparator logic in the second high-density PLD. The search state machine began by presenting to the TRT the highest order address: A12 in a 13-bit address series, A0 being the lowest order. This address indexed into the midpoint of the TRT, essentially cutting the 8K table in half. The data indexed by the state machine appeared at the comparator input port, where was compared with the TR address data. The comparator reported the address as less than or equal to the TRT address, using the assumption that if it's not less than or equal to, it must be greater than. If equal to, then the packet was local and discarded, and the state machine maintained the same address for the next comparison. If less than or greater than, the state machine then reset the next address bit in series—which bisects either the upper or lower 4K memory segment—and presented it to the TRT for the next cycle.

NETBuilder II Token-Ring Module 96 Bit Latch & Comparator Block Diagram

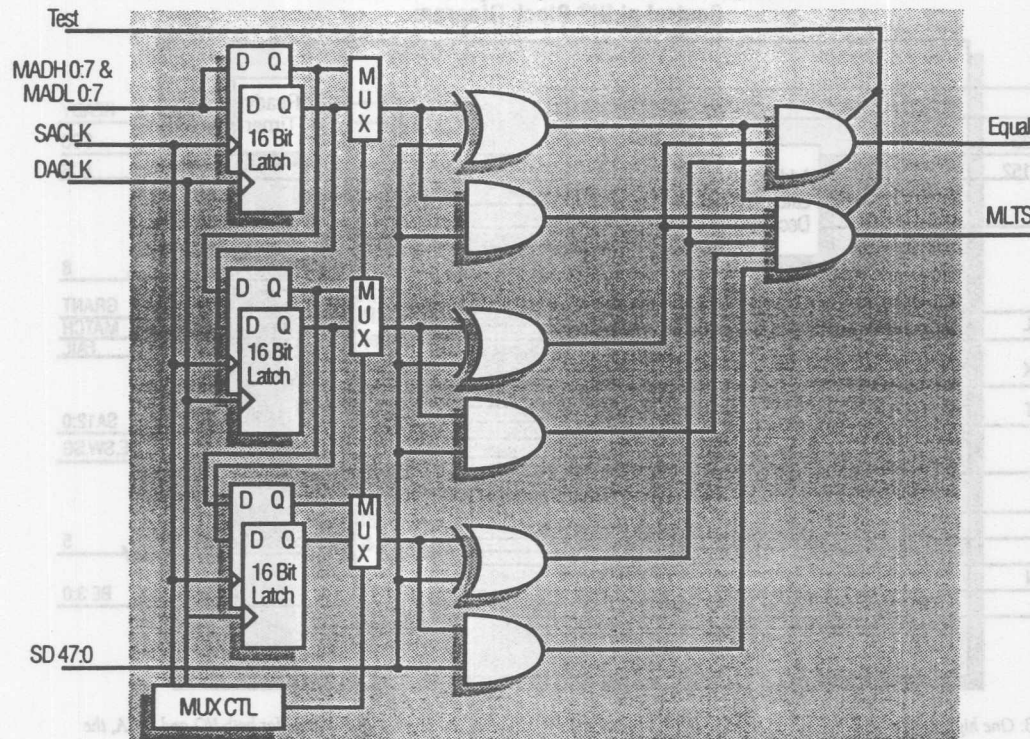


Figure 4: The second high-density PLD (pLSI 1048) holds the two 48-bit latches and the 48-bit comparator for the Binary Search state machine.

The process repeated until the machine ran out of addresses or found a match. If no match was found, then the packet was not local and became forwarded to the Am29K (and on to the bridge). Each address step (13 in all) had to take place within the 80 nsec cycle time, including the RAM access time and the setup and hold time of the parts forming the comparator and search state machine.

The goal, due to the small amount of real estate, was to use one high-density PLD to hold all of the random logic including I/O select, buffer control for both I/O and DMA, the TRT page register, and the binary search state machine address generator (see Figure 3). A second high-density PLD would hold the two 48-bit latches and the 48-bit comparator (Figure 4).

Simply using databook figures for gate delay proved to be too cumbersome and inaccurate

for evaluating high-density PLDs. Instead, I built models for the binary search address generator state machine and the 48-bit address latch and comparator. I then modified the models to work within the architectures and design environments of each logic device. The logic would be implemented in the device. The resulting structural implementation would then be analyzed for timing, with a known number of logic levels producing estimated path delays through the circuit. The physical implementations also ensured that the required density could be implemented within the devices.

Initial calculations for the Actel FPGAs ruled them out immediately. Then I tuned to the newly introduced MACH devices from AMD. Working with AMD support personnel, I was able to create a version of the algorithm that would work within the devices. While appearing to meet density goals, I found that the

MACH devices couldn't meet the worst-case 80-nsec timing requirement. In fact, it appeared it would take them two clock cycles—160 nsec—to implement the function.

At this time Lattice Semiconductor was also introducing high-density logic devices—the pLSI family. Having used Lattice GAL devices for some time, I began working with Lattice to evaluate the pLSI devices. The difficulty here was the initial software support for the pLSI devices required me to rewrite my equations from a high-level ABEL format to lower-level Boolean equations. Another trick was learning how to map the design into the resources of the pLSI devices—circuits called Generic Logic Blocks (GLBs). Specifically, I had to partition my implementation to match the four-register and four-output GLB architecture. Newer versions of the pLSI software support these activities directly.

With the design rewritten and partitioned, I compiled the design into the pLSI architecture. It became clear that the largest pLSI device, the pLSI 1048, would be able to accommodate all the logic I needed and still meet the 80 nsec cycle time as well.

Building prototypes

At the time of the first prototype board the pLSI 1048 was not yet available. The smaller pLSI 1032 was available, so I used two of them for development purposes. Since the full 48-bit latch and compare function wouldn't fit on the pLSI 1032, I decided to latch and search only go destination addresses in the comparator pLSI device. This was sufficient to prove out all the control logic and the comparator from a hardware perspective.

Due to the short design cycle and the fact that the logic was not fully complete (meant to be designed incrementally, in fact), a problem occurred with the locking of signals to specific pLSI pins. This problem was overcome with an 84-pin wire-wrap socket from Emulation Technology, so changes in pin assignments would not affect board layout. The second version of the prototype with the wire-wrap socket was produced in small quantity for software and QA teams to start work in parallel with my efforts to complete a final version of the board. Due in part to the efforts described here, new versions of the Lattice Semiconductor software have

corrected this pin assignment problem.

With availability of the pLSI1048, a third and final prototype board was generated correcting all known logic problems. This board supported destination and source address checking functions as well as many other required features.

This Token Ring adapter module is now shipping in the NETBuilder II bridge/routers. It has successfully expanded the series to new customer applications, and 3COM's competitiveness, to the very important Token Ring segment of the market for Intermediate Systems.

A Decision Process Used for FPGA Selection in Digital Signal Processing for Fiber Optic Sensors

This paper was presented at the 1993 PLD Design Conference and Exhibit in Santa Clara, California.

A Decision Process Used for FPGA Selection in Digital Signal Processing for Fiber Optic Sensors

Beck Mason, Vice President of Engineering

FiberMetrics Corporation

Introduction

In 1993, if one wants to incorporate programmable logic devices into a design, there is no shortage of possibilities—everything from one-time programmable bipolar parts to electronically erasable (EE) CMOS devices. Yet PLDs, especially when used to consolidate existing discrete functions, can provide many tangible benefits. Using a PLD can reduce board utilization and electromagnetic interference, improve design reliability, simplify manufacturing, and streamline parts ordering and inventory processes.

The abundance of devices at varying levels of complexity has created many PLD choices. Yet all these possibilities can make selecting the right device extremely difficult. In its search for a PLD, FiberMetrics gained some useful insight into choosing both a device and manufacturer. This paper describes the technical, organizational, and economic issues that FiberMetrics considered in its search. But the major lessons the company learned are simple: success depends on how well you understand your application, and how well you understand your costs.

Application description

FiberMetrics is a small company that develops a variety of fiber-optic sensor systems. Used primarily in the research and development sectors of the aerospace industry, these systems measure strain and temperature in composite material structures.

Each fiber-optic sensor is an interferometric device that produces a periodic nonlinear response function. A change in temperature or strain causes a shift in the sensor's interference pattern. However the sinusoidal nature of this pattern makes it difficult to calculate strain or temperature exactly. So the sensor system uses a tunable semiconductor laser diode to launch light into the optical fiber (Figure 1). The system then rapidly shifts the diode's wavelength, causing a change in the interference pattern. By sweeping over a fringe and measuring its phase electronically, the system compares the resultant phase shift with the previous interference pattern, then generates a strain or temperature measurement.

At the heart of the sensor system, and the target of FiberMetrics device consolidation effort, is a phase demodulation board (Figure 2). This board receives

FIGURE 1

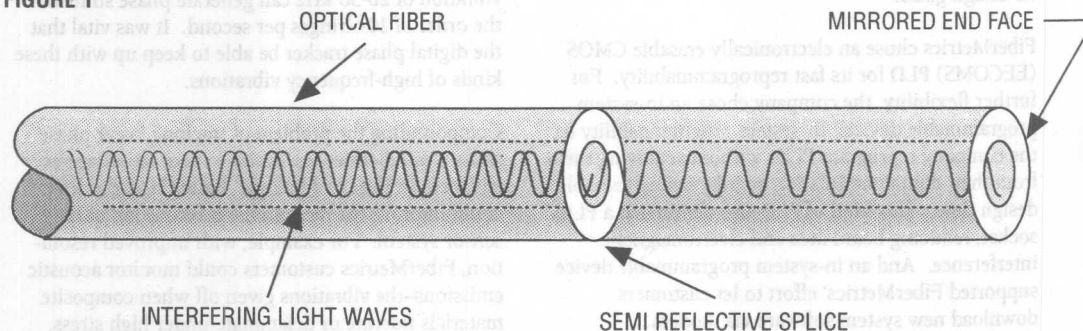


Figure 1, Fiber-Optic Sensor: A FiberMetrics sensor is an interferometric device whose interference pattern shifts with changes in temperature or strain. Because the sinusoidal nature of these patterns make precise calculations difficult, the sensor system uses a tunable semiconductor laser diode to launch light into an optical fiber, then rapidly shifts the diode's wavelength, causing a change in the interference pattern. By sweeping over a fringe and measuring its phase electronically, the system compares the resultant phase shift with the previous interference pattern and generates a strain or temperature measurement.

an analog input signal, digitizes it, then passes the signal on to a digital phase tracking system. The phase tracker then outputs strain or temperature information to a 16-bit analog-to-digital converter and a multiplexed 16-bit data bus. There are four phase demodulation boards in a FiberMetrics sensor system, each with a digital phase tracker. Originally composed of fourteen discrete logic components, FiberMetrics reduced each digital phase tracker to one complex PLD (CPLD) (Figure 3).

Comparing CMOS, fuse, and anti-fuse technologies

When considering programmable versus non-programmable technologies, some of the more important questions to consider are the kinds of design changes you anticipate, their frequency, and whether you have the design volume to afford any production waste or outdated inventory.

Because of their relatively high cost, FiberMetrics produced a small number of sensor systems—30 to 40 per year. Furthermore, the digital phase tracking algorithms in these systems varied slightly, depending on the application (i.e. temperature or strain sensing). The company's goal was to create a system that it could upgrade as it realized system performance advances. In fact, the company's plans called for customers to eventually be able to customize their own sensor systems. Because all these needs placed a premium on reusability and reprogrammability, FiberMetrics concluded that fuse and anti-fuse technologies would not enable the company to meet its design goals.

FiberMetrics chose an electronically erasable CMOS (EECOMS) PLD for its fast reprogrammability. For further flexibility, the company chose an in-system programmable device. In-system programmability let the company reprogram PLDs without removing them from their digital modulation boards, saving valuable design time. This kind of PLD also eliminated a PLCC socket, reducing board area and electromagnetic interference. And an in-system programmable device supported FiberMetrics' effort to let customers download new system software via modem.

In addition to reprogrammability, comparing CMOS to fuse and anti-fuse devices raised the issues of power consumption and dissipation. CMOS runs cooler and consumes less power than other PLD technologies; the FiberMetrics design required both these attributes. First, many of the company's systems are used in the

aerospace industry, an industry that considers minimizing design size and power consumption absolutely essential. Second, because of the sensor system's size restrictions, digital phase trackers were packed close to temperature-sensitive parts such as laser diodes, making low heat dissipation essential. Finally, because using programmable logic to replace discrete components tends to dramatically increase power consumption, it was critical to minimize that effect. Using CMOS technology, FiberMetrics constructed a PLD-based sensor system whose total power draw was only 13% more than its original, discrete logic design.

Evaluating speed

Unfortunately, the different ways in which PLD manufacturers specify maximum device speed make it difficult to evaluate potential candidates. One consistent, understandable measurement that FiberMetrics came to rely upon was gate delay. While peak clock frequency was also important, the company was less certain of the calculations behind this specification. Consequently, gate delay became the preeminent speed benchmark.

Speed was critical to the success of the FiberMetrics system, especially when it came to tracking the phase shifts of its interferometric sensors. The system's tracking speed is directly related to the level of strain and temperature it can monitor. Yet high-frequency vibrations can generate phase shifts that exceed a system's tracking capabilities—even a small amplitude vibration of 20-30 kHz can generate phase shifts on the order of 10^5 fringes per second. It was vital that the digital phase tracker be able to keep up with these kinds of high-frequency vibrations.

Compounding the problem of tracking faster phase shifts was FiberMetrics' desire to take advantage of advances in laser technology. Faster laser diodes could improve the measurement resolution of the sensor system. For example, with improved resolution, FiberMetrics customers could monitor acoustic emissions—the vibrations given off when composite materials fracture or delaminate under high stress. But using faster diodes also causes faster phase-shift variations, thus increasing the need for even more system speed.

To accommodate its multiple speed requirements, FiberMetrics specified its PLD-based phase tracker at 33 MHz. However, the company chose a device with

Total resources included 32 generic logic blocks (GLB) and 64 I/O blocks	
Eight bit D flip-flop	2 GLB
Eight bit Latch	2 GLB
Sixteen bit D flip-flop with asynchronous clear	4 GLB
Eight bit full adder with carry	4 GLB
Sixteen bit up/down counter with preload	5 GLB
Two D flip-flops	1 GLB
Two four bit magnitude comparators	1 GLB
Two bit D flip-flop	1 GLB
Eight bit D flip-flop (input)	8 I/O
Sixteen bit tri-state output buffer	16 I/O
Eight bit magnitude comparator	1 GLB
And an Assortment of other logic and input and outputs	

Figure 2, Phase Demodulation Board: The FiberMetrics sensor system contains four phase demodulation boards. These boards receive analog input signals, digitize them, then pass them on to the digital phase trackers that, depending on the application, will output strain or temperature information.

an upper speed boundary of 80 MHz to accommodate future technology improvements.

In addition to actual system speed, propagation delay can also be an important criterion when selecting a PLD.

For example, in designs where synchronization plays an important role, a consistent propagation delay may be absolutely mandatory. The phase tracking circuits of the FiberMetrics sensor systems are synchronized with the wavelength modulation of their laser diodes. The circuits generate synchronized timing signals that control the sampling of the sensor signal, and the phase demodulation. Consequently, any signal lag or timing structure change could affect the system's phase measurement accuracy. Additionally, the digital tracker's operation is completely asynchronous, making it very susceptible to timing delay variations.

Beyond these concerns, however, it was system resolution that most affected FiberMetrics' propagation delay deliberations. To illustrate: the system detected an optical signal from its interferometric sensor with a photodiode, amplified the signal, and then sampled it using an eight-bit flash analog-to-digital converter. The system's phase resolution depended on the speed and accuracy with which could sample the optical signal. And in order to

achieve a phase resolution target of 1.5°, FiberMetrics calculated that its PLD must have a total propagation delay of less than 20 ns.

Evaluating density

Because device structures and utilization schemes vary from manufacturer to manufacturer, FiberMetrics found density claims as hard to evaluate as speed specifications. The company concluded that the key to evaluating device density was understanding an application's functions.

To calculate a prospective design's density, consider making a list of its major functions and their gate equivalents. Then carefully examine the logic structures of your PLD candidates to see how well these functions fit. It may also be helpful to consult device manufacturers as to a PLD's best use. For instance, a device may be better suited as a state-machine or high-speed counter than as a shift register or register file.

Software and software tool vendors are another useful source for density information. A vendor that supports a wide range of devices may be able to offer valuable insight into the PLD you are considering, perhaps even refer you to a customer already using the device. You could also purchase development tools that support the device under consideration and experiment, using the appropriate simulation modules to test the applications. Or you could buy Open ABLE-compatible tools, generate JEDEC files, port the files to a variety of devices, then use the appropriate simulation tools to compare performance and utilization. Regardless, making a small software investment can pay large dividends—especially when manufacturing large numbers of products—because it gets you closer to finding the most appropriate PLD.

Ironically, FiberMetrics had no specific density goal for its PLD-based circuit. Because the PLD had to accommodate future software upgrades, the company created a generic logic block equivalent-list of its major functions (Figure 4), then calculated the minimum number of gates necessary to implement the existing logic (4000). As it evaluated potential candidates, FiberMetrics closely examined the abilities of manufacturers to provide a device growth path. The company's most important task, however,

FIGURE 2

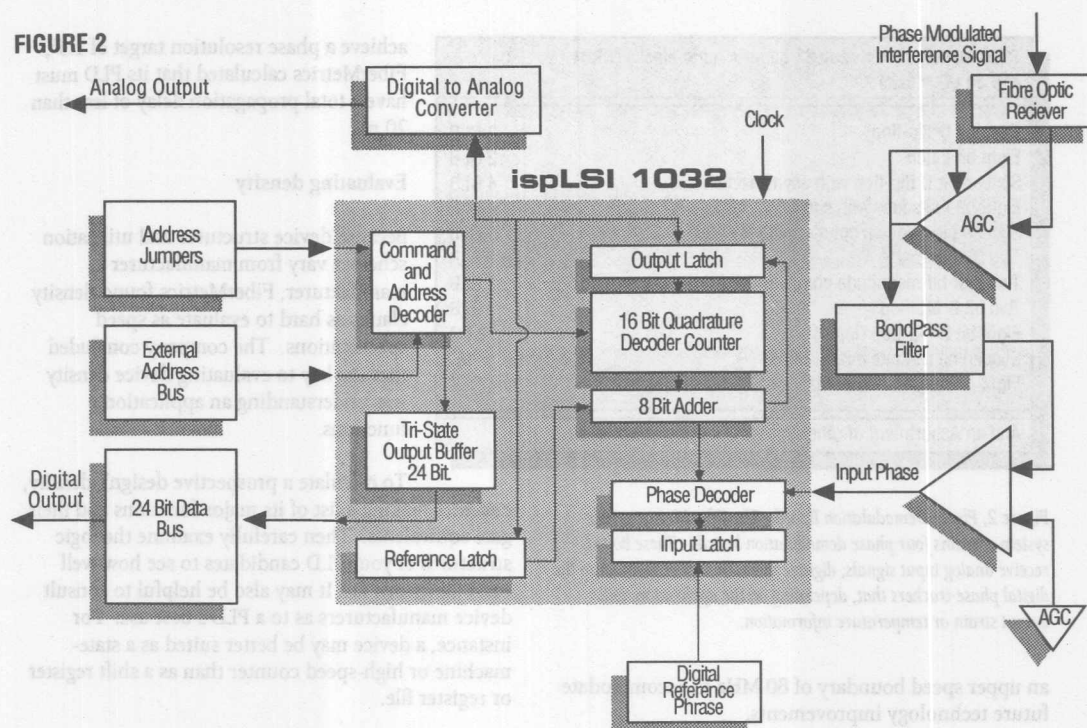


Figure 3, Digital Phase Tracking Circuit: Using analog input, the FiberMetrics digital phase trackers output strain or temperature information. Originally comprised of fourteen discrete logic components, the company consolidated each tracker into one complex PLD (CPLD)-the ispLSI 1032 PLD from Lattice Semiconductor.

remained finding the right PLD for its application.

The design of the digital phase tracker emphasized data transfer over manipulation: latching a phase signal reference, the tracker compared that reference with a digitized photodetector signal, performed a phase comparison, then generated a high-resolution phase byte plus a control signal, and sent the signal to a 16-bit up/down counter and a multiplexed 16-bit data bus. Because the latching and data manipulation in latches were register-intensive operations, the company placed a premium on control logic and input/output pins. And because of the small amount of manipulation outside the latches, FiberMetrics put less emphasis on product terms. The company's final choice was a more general-purpose PLD, though one well-suited for interfacing with microprocessors. The PLD had 192 total registers, 64 I/O pins, input blocks that could latch data, and tri-state outputs that the company could convert into I/O pins—all necessary characteristics for successfully implementing the digital phase tracker.

Software development tools

Given the potential complexity of a PLD design and the shrinking time-to-market windows, easy-to-use software development tools are of paramount importance.

Development tools, at a minimum, should offer Boolean logic entry. Schematic capture is even better, as it lets engineers visualize design flow and recognize errors quickly. For example, if a designer improperly assigns a variable using Boolean Algebra, that error may not be readily apparent. However, if a designer connects a trace incorrectly using schematic capture, the problem much easier to detect.

Library functions are also helpful for rapid design development. Existing functions provide a foundation from which you can build, either by editing the descriptions or by using them as programmatic examples for functions you want to create. Not surprisingly, the litmus test for any good library is that it should never be harder to enter a function automatically than to create one by hand.

When developing its PLD logic, FiberMetrics made extensive use of library functions to reduce the learning curve of its software development tools and PLD. In addition, the tools displayed conceptual representations of internal PLD logic, letting designers view the specific logic blocks they were using as they entered equations and systems. Not only did this display help designers visualize available resources, it helped them to more easily manage the overall utilization of their design.

While ease of use is essential, affordable software development tools are nothing to be overlooked. Low-priced tools can help you evaluate potential hardware and software without making a major investment. A trial period with a particular tool set may even influence your purchasing decision—after all, if a PLD meets your technical specifications, and your designers are already familiar with its software tools,

why switch to another manufacturer and endure a new hardware and software learning curves?

The price of FiberMetrics software tools did not dictate its device selection—but tool cost did accelerate the company's development schedule. The overall price of these development tools was less than \$5000. With a software tool price that did not require a significant financial commitment, and a device that met all the company's technical specifications, FiberMetrics decided to begin its product design cycle two to three months ahead of schedule.

It is important to reiterate that software price should never be the linchpin on which the PLD selection process hinges. But based on FiberMetrics' experience, the affordability of software development tools is a factor that merits serious consideration.

FIGURE 3

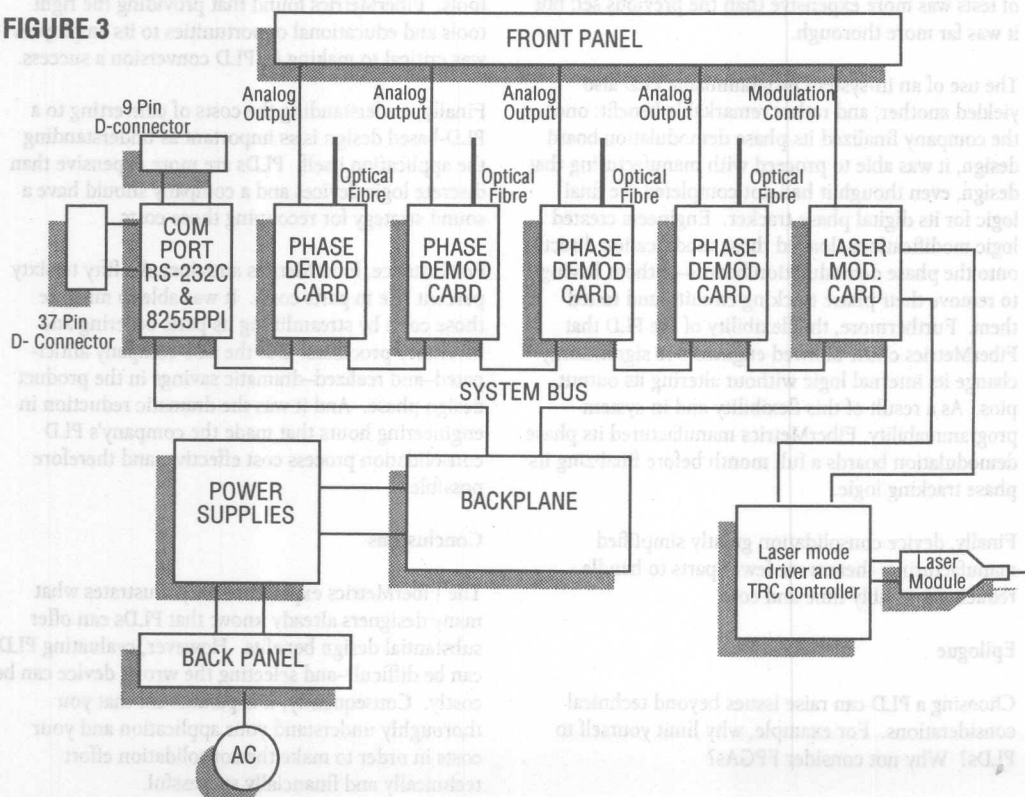


Figure 4, Major Functions List: To calculate design density, consider making a list of your major functions and their gate or generic logic block equivalents. Then carefully examine the logic structures of your PLD candidates to see how well the functions fit. From the above list, FiberMetrics calculated the minimum number of gates necessary to implement its digital phase tracking logic then chose a PLD on the basis of how well it implemented that logic.

Measurements of success

By consolidating discrete components into one CPLD, FiberMetrics realized several important benefits. First, when compared to the original phase demodulation board, the PLD-based digital phase tracker cut design development and board rework time by seventy-five percent. The company also completed its new design using just one board set, instead of two sets it previously required.

Additionally, using an electronically erasable PLD streamlined device testing. The rapid reprogrammability of EE technology let FiberMetrics designers isolate problematic device areas, test them, and implement logic changes quickly. With the time they saved, the designers were able to execute a more robust testing suite, including high-speed tests for output filters and the generation of non-realistic outputs to check system response. This new battery of tests was more expensive than the previous set; but it was far more thorough.

The use of an in-system programmable PLD also yielded another, and rather remarkable benefit: once the company finalized its phase demodulation board design, it was able to proceed with manufacturing that design, even though it had not completed the final logic for its digital phase tracker. Engineers created logic modifications, loaded those modification directly onto the phase demodulation boards—without having to remove their phase tracking circuits—and tested them. Furthermore, the flexibility of the PLD that FiberMetrics chose allowed engineers to significantly change its internal logic without altering its output pins. As a result of this flexibility and in-system programmability, FiberMetrics manufactured its phase demodulation boards a full month before finalizing its phase tracking logic.

Finally, device consolidation greatly simplified manufacturing: there were fewer parts to handle, reducing assembly time and cost.

Epilogue

Choosing a PLD can raise issues beyond technical considerations. For example, why limit yourself to PLDs? Why not consider FPGAs?

While there are similarities between CPLDs and FPGAs, FiberMetrics found several significant differences that went to the heart of its digital phase

tracking application. First, the tracker's operation—which emphasized data transfer over manipulation—placed a premium on control logic. PLDs offered a more efficient implementation of control logic than did FPGAs. Second, the synchronized nature of the sensor system and the need for precise signal sampling made predictable propagation delays mandatory. Coupled with the fact that the system's software was rapidly evolving, FiberMetrics found it could not rely on FPGAs to provide consistent propagation delays throughout the life of the design. Finally, the company felt that CPLDs, with their individual logic blocks, were simply conceptually easier to program than FPGAs.

When converting a design from discrete devices to a PLD, a company may also confront the need for organizational changes. It will probably be necessary to send design and production engineers to educational classes to learn about a device and its software tools. FiberMetrics found that providing the right tools and educational opportunities to its employees was critical to making its PLD conversion a success.

Finally, understanding the costs of converting to a PLD-based design is as important as understanding the application itself. PLDs are more expensive than discrete logic devices and a company should have a sound strategy for recouping those costs.

For instance, FiberMetrics anticipated a fifty to sixty percent rise in parts costs. It was able to mitigate those costs by streamlining its parts ordering and inventory processes. But the also company anticipated—and realized—dramatic savings in the product design phase. And it was the dramatic reduction in engineering hours that made the company's PLD consolidation process cost effective, and therefore possible.

Conclusions

The FiberMetrics experience only illustrates what many designers already know: that PLDs can offer substantial design benefits. However, evaluating PLDs can be difficult—and selecting the wrong device can be costly. Consequently, it is paramount that you thoroughly understand your application and your costs in order to make the consolidation effort technically and financially successful.

Learn the Fundamentals of Digital Filter Design

This article is reprinted from *Electronic Design* — July 25, 1991.

Historically, designers often have taken an analog approach to filtering. Filters were constructed using operational amplifiers, resistors, and capacitors. One approach would implement a second-order filter, and higher-order filters could be implemented by cascading second-order filters. However, passive components with tolerances of 1% or better are necessary for the filter to have reproducible characteristics. And the filter is typically fine-tuned by trial-and-error substitution of available component values. In addition, operational amplifiers with a high gain-bandwidth product may be needed to keep undesirable phase shift to a minimum or keep a closed-loop system stable. These factors are among the many problems in real-world implementations of filters.

With the advances made in digital signal processing, however, digital filters are becoming a more attractive design alternative to traditional analog techniques. Because digital system information is in digital form, filtering can be accomplished relatively easily by passing the data through a filter algorithm. In addition, digital filters have the advantages of no filter characteristic drift over time, temperature, or voltage. And they can easily be designed to filter low-frequency signals. Moreover, the filter response can be made to closely approximate the ideal response, and linear phase characteristics are possible.

There are many well-established methods of determining the filtering algorithm. Basically, the designer establishes the desired filter characteristics, then by yielding a filter transfer function. The continuous-time transfer function is then transformed to the equivalent linear discrete time-difference function. This function in the Z domain has the general form of:

$$G(Z) = (A_0 + A_1 Z^{-1} + A_2 Z^{-2} + \dots + A_N Z^{-N}) / (B_0 + B_1 Z^{-1} + B_2 Z^{-2} + \dots + B_M Z^{-M}) \quad (1)$$

The equation is referred to as the pulse transfer function. It's actually the Z transform of the continuous-time filter's unit impulse response. Conversely, the inverse Z transform of the pulse transfer function yields the impulse response of the filter.

The coefficients A_n and B_m determine the response of the digital filter. Changing

WIRETECH
Intellitec Semiconductor Corp., Carlsbad Pacific Center One, 101 Phoenix Avenue Rd.,
Third Floor, Carlsbad, CA 92008 (619) 331-4121

BASIC TECHNIQUES LET DESIGNERS
BUILD A FINITE-IMPULSE-RESPONSE
FILTER IN DEDICATED HARDWARE
USING PROGRAMMABLE LOGIC.

LEARN THE FUNDAMENTALS OF DIGITAL FILTER DESIGN

Historically, designers often have taken an analog approach to filtering. Filters were constructed using operational amplifiers, resistors, and capacitors. One op amp could implement a second-order filter, and higher-order filters could be implemented by cascading second-order filters. However, passive components with tolerances of 1% or better are necessary for the filter to have reproducible characteristics. And the filter is typically fine-tuned by trial-and-error substitution of available component values. In addition, operational amplifiers with a high gain-bandwidth product may be needed to keep undesirable phase shift to a minimum or keep a closed-loop system stable. These factors are among the many problems in real-world implementations of filters.

With the advances made in digital-signal processing, however, digital filters are becoming a more attractive design alternative to traditional analog techniques. Because digital-system information is in digital form, filtering can be accomplished relatively easily by passing the data through a filter algorithm. In addition, digital filters have the advantages of no filter-characteristic drift over time, temperature, or voltage. And they can easily be designed to filter low-frequency signals. Moreover, the filter response can be made to closely approximate the ideal response, and linear phase characteristics are possible.

There are many well established methods of determining the filtering algorithm. Basically, the designer establishes the desired filter characteristics, thereby yielding a filter transfer function. The continuous-time transfer function is then transformed to the equivalent linear discrete-time-difference function. This function in the Z domain has the general form of:

$$G(Z) = (A_0 + A_1 Z^{-1} + A_2 Z^{-2} + \dots + A_n Z^{-n}) / (1 + B_1 Z^{-1} + B_2 Z^{-2} + \dots + B_m Z^{-m}) = Y(Z)/X(Z)$$

The equation is referred to as the pulse transfer function. It's actually the Z transform of the continuous-time filter's unit impulse response. Conversely, the inverse Z transform of the pulse transfer function yields the impulse response of the filter.

The coefficients A_n and B_m determine the response of the digital filter. Changing

MIKE TRAPP

Lattice Semiconductor Corp., Carlsbad Pacific Center One, 701 Palomar Airport Rd., Third Floor, Carlsbad, CA 92009; (619) 931-4751.

DIGITAL FILTERS

the coefficients changes the response of the filter. The terms Z^{-n} and Z^{-m} represent sampling delays or taps. The $G(Z)$ equation represents the algorithm of sampling the input, multiplying it by A_0 , and adding it to the previous sample that's been multiplied by A_1 , then adding that value to the next previous sample which has been multiplied by A_2 , and so on. An output value occurs when all N values have been multiplied and accumulated.

In parallel, each output value is stored, multiplied by B_1 , then added to the previous output value which has been multiplied B_2 , and so on. The equation can be rearranged so that the result of the output multiply accumulate is added to the result of the input multiply accumulate to produce an output. This procedure is referred to as convolution. An output sample is produced for every input sample (Fig. 1).

The key to digital-filter design is to determine the filter coefficients that will produce the desired frequency response. Recursive digital filters, or infinite-impulse-responsive (IIR) filters, are a type of digital filter in which the design methodology closely follows that of an analog filter. One method for determining the coefficients is to define a realizable

continuous-time domain Chebyshev, Butterworth, or equal-ripple filter then use Z transforms to transform the continuous-time-domain transfer function to the equivalent discrete-time transfer function that yields the filter coefficients.

A second popular method is the bilinear transform. In this method, engineers first design an analog filter so that after it's transformed to a digital filter, the resulting filter meets a set of desired digital-filter specifications. This analog filter is then transformed to a digital filter via the bilinear transform from the S variable of the Laplace transform to the Z variable of the Z transform.

In a non-recursive digital filter or finite-impulse-response (FIR) filter, the output is computed using the present input X_n and the previous inputs $X_{n-1}, X_{n-2} \dots X_{n-N}$. This implies that the coefficients, B_m , are all 0, and there's no feedback from the output. Designing non-recursive digital filters (FIR) involves defining an ideal desired frequency response from which the ideal impulse response is computed. The ideal impulse response is truncated to a finite number of non-zero samples using a windowing function, which is judiciously chosen. A common windowing function is the Kaiser window function.

An interesting property of FIR filters is that if an FIR system has linear phase, then its frequency response is constrained to be zero at $f = 1/2T$, where T equals the sampling frequency if:

$$h[M - n] = h[n] \text{ and } M \text{ is odd. (} M = \text{truncation length of the window).}$$

This implies the M should be even when designing high-pass and band-stop filters. Or,

$$h[M - n] = -h[n] \text{ and } M \text{ is even.}$$

A second method is the Parks-McClellan method. In this approach, the filter order and the edges of the passbands and stopbands are fixed, and the impulse-response coefficients are varied systematically so that an equal-ripple behavior is achieved in each approximation band. With this approach, the filter order can't be specified in advance. Therefore, a cut and try procedure must be used to find the minimum filter order. The cut and try can be reduced by using a formula that predicts the filter order required to meet a given set of specifications.

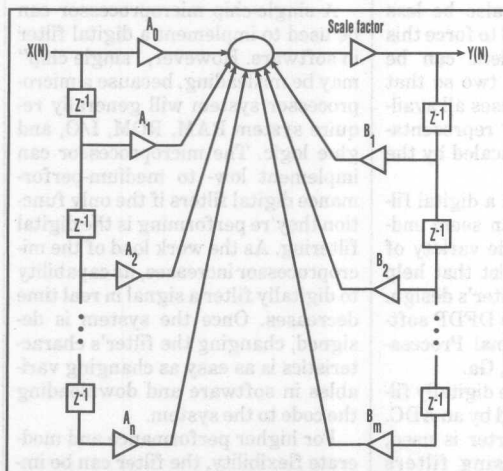
There are advantages and disadvantages to each type of digital filter (IIR and FIR). An FIR filter is always stable because there's no feedback from the output and the impulse response is finite. In addition, the amplitude and phase can be arbitrarily specified. On the other hand, an FIR filter will generally require more taps, and consequently more math, to compute the output value. The design methodology doesn't resemble the familiar analog design techniques.

An IIR will generally have fewer coefficients, but the required output feedback can make circuit implementation more complex. A stable IIR filter can become unstable if the coefficients aren't chosen properly to account for digital math errors.

There are four main types of errors that can arise in the design of digital filters. These are referred to as quantization errors. They are:

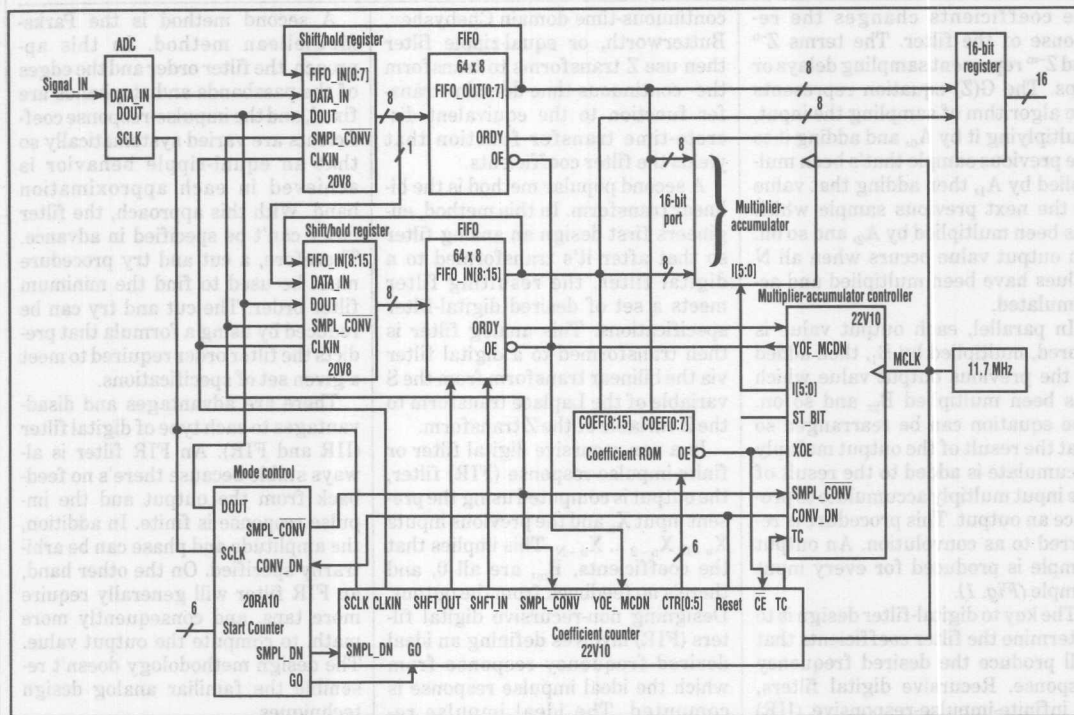
1. Quantization errors of the input analog-to-digital conversion
2. Quantization errors of the coefficients
3. Quantization errors due to arithmetic computations, including overflow
4. Limit cycles

In most cases, a 12-bit analog-to-digital converter (ADC) provides enough dynamic range and sufficiently small quantization noise. If floating-point numbers are used for the filter coefficients, the quantization error is usually small enough. However, floating-point arithmetic is more complex and more expensive



1. IN THE FUNCTIONAL structure of a digital filter, the A and B coefficients determine the response of the filter and the Z terms represent sampling delays called taps.

DESIGN APPLICATIONS DIGITAL FILTERS



2. AN FIR FILTER IS IMPLEMENTED in a circuit that uses a single-port 16-bit multiplier-accumulator capable of a 85-ns clock speed. Because it's based on microcode, the multiplier-accumulator can be controlled with a PLD.

to implement than integer or fixed-point arithmetic. If 12- or 16-bit coefficient are used, the quantization error is generally negligible.

In the digital domain, math is performed using finite precision binary arithmetic. All digital filters need to multiply a signal sample by a constant coefficient. Of course, multiplying 2 N-bit binary numbers results in a 2N-bit result, but digital systems are usually confined to a fixed number of bits with which to represent binary numbers. Therefore, it's necessary to round off the 2N-bit digital number back to N bits. If a 32-bit multiply accumulator is used and the final output is rounded to 16 bits, the arithmetic quantization errors can be minimized.

If overflow occurs during mathematical operations, the digital filter can behave in a nonlinear fashion and oscillations can occur. Twos-complement arithmetic can help eliminate overflow. In addition, a satu-

rating adder can be used. If the coefficients are less than one, then the resulting product will also be less than one. Scaling is used to force this condition. The coefficient can be scaled by a multiple of two so that the largest coefficient uses all available bits in the binary representation. The input is then scaled by the same amount.

The detail with which a digital filter can be described can seem endless. Fortunately, a wide variety of computer programs exist that help the engineer with the filter's design. One such product is the DFDP software from Atlanta Signal Processing Inc. (ASPI), Atlanta, Ga.

Before a signal can be digitally filtered it must be digitized by an ADC. If a delta-sigma converter is used, the need for antialiasing filters (which must be analog and can be many orders) is virtually eliminated. Delta-sigma converters may have sample rates as high as 100 kHz. The

filter algorithm can then be implemented in software or hardware.

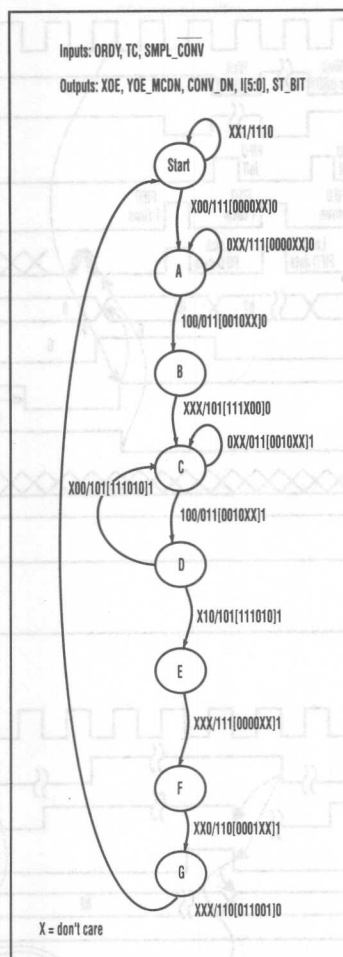
A single-chip microprocessor can be used to implement a digital filter in software. However, "single chip" may be misleading, because a microprocessor system will generally require system RAM, ROM, I/O, and glue logic. The microprocessor can implement low- to medium-performance digital filters if the only function they're performing is the digital filtering. As the work load of the microprocessor increases, its capability to digitally filter a signal in real time decreases. Once the system is designed, changing the filter's characteristics is as easy as changing variables in software and downloading the code to the system.

For higher performance and moderate flexibility, the filter can be implemented in dedicated hardware using programmable logic for design flexibility. The limiting parameter will be the time to do a multiply-accu-

mulate function and the amount of physical space required for the hardware implementation of the taps. Consider a circuit that uses a single-port 16-bit multiplier-accumulator capable of an 85-ns clock speed (Fig. 2). The device can work in twos-complement numbers and has output saturation capabilities. As stated before, these two features are desirable when implementing digital filters. In addition, the device can be easily controlled with a programmable logic device (PLD) because it's microcoded based.

First, the system must initially load the first N (N = 64) samples into the FIFO before any convolution takes place. Otherwise, the FIFO would never fill up. A counter implemented in a 20RA10 works well. The 6-bit counter is implemented with the four least-significant bits implemented as an asynchronous counter. SMPL_DN (ADC sample done) acts as the clock. The two most-significant bits are implemented as a ripple counter. This type of counter design makes it possible for a long counter to be implemented with only four product terms per output. The SMPL_DN signal is also generated in the 20RA10, and is triggered off signals from the ADC.

When the counter reaches the value 63, indicating that the FIFO is full minus the one sample that's held in the shift/hold register, GO becomes true and the system begins to execute the filtering algorithm. Because the system is linking two asynchronous subsystems (ADC and the multiplier-accumulator), there must be an asynchronous interface between the two. The 20RA10 is utilized by generating one interface signal SMPL_CONV (sample or convolve mode). The system powers up with this line held in the sample mode (SMPL_CONV = 1). When GO goes true, synchronous with the falling edge of the clock from the ADC, SMPL_CONV goes low asynchronously with MCLK (synchronous with SCLK). Because SMPL_CONV is an input to the state machine, the machine could be subject to a metastable input. The Lattice CMOS PLDs are very high



3. AN 8-STATE state machine implements the operations of loading a sample into the multiplier-accumulator, then loading the coefficients in and issuing the multiply-accumulate command until all N samples are done.

speed, so the metastable characteristics are excellent. That is, the state flip-flop has a very low probability of going metastable. Therefore, the state machine will have to wait, at most, one extra MCLK cycle before starting the convolution.

Once the convolution is started, the operations of loading a sample into the multiplier-accumulator, then loading the coefficient into the

multiplier-accumulator and issuing the multiply-accumulate command, can be repeated until all N samples have been done. At this time, the filter output is valid and the cycle is restarted. These steps can be implemented with an 8-state state machine (multiplier-accumulator controller) (Fig. 3).

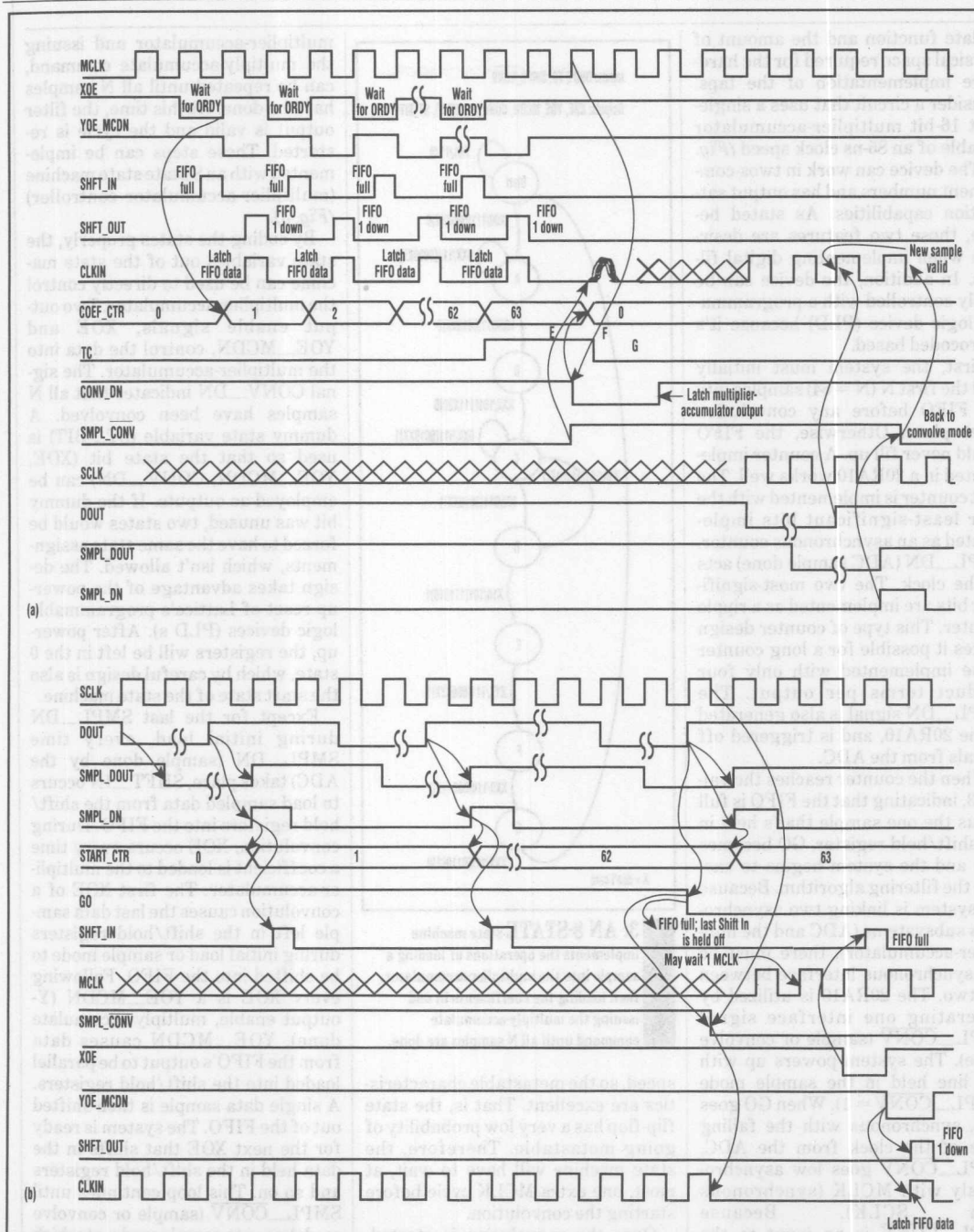
By coding the states properly, the state variables out of the state machine can be used to directly control the multiplier-accumulator. Two output enable signals, XOE and YOE_MCDN, control the data into the multiplier-accumulator. The signal CONV_DN indicates that all N samples have been convolved. A dummy state variable (ST_BIT) is used so that the state bit (XOE, YOE_MCDN, CONV_DN) can be employed as outputs. If the dummy bit was unused, two states would be forced to have the same state assignments, which isn't allowed. The design takes advantage of the power-up reset of Lattice's programmable logic devices (PLDs). After power-up, the registers will be left in the 0 state, which by careful design is also the start state of the state machine.

Except for the last SMPL_DN during initial load, every time SMPL_DN (sample done by the ADC) takes place, SHFT_IN occurs to load sampled data from the shift/hold registers into the FIFO. During convolution, XOE occurs every time a coefficient is loaded to the multiplier-accumulator. The first XOE of a convolution causes the last data sample left in the shift/hold registers during initial load or sample mode to be shifted into the FIFO. Following every XOE is a YOE_MCDN (Y-output enable, multiply-accumulate done). YOE_MCDN causes data from the FIFO's output to be parallel loaded into the shift/hold registers. A single data sample is then shifted out of the FIFO. The system is ready for the next XOE that shifts in the data held in the shift/hold registers and so on. This loop continues until SMPL_CONV (sample or convolve mode) goes to sample mode, at which time a new sample is loaded into the shift register, restarting the cycle.

Inputs to the state machine,

DESIGN APPLICATIONS

DIGITAL FILTERS



4. FIFO CONTROL SIGNALS are generated asynchronously. The system timing diagrams for the convolve (a) and initial load (b) operations show the appropriate Shift In and Shift Out signals, and clock signals sent to the shift/hold register.

SMPL_CONV, tell the machine when it's time to begin the convolution cycle. This signal comes from the mode-control device. TC (Terminal Count) indicates when the convolution is to end. TC comes from a 6-bit coefficient counter, and is valid when the count equals 63, which indicates when all 64 samples have been convolved with the respective coefficients. ORDY comes from the FIFO and tells the state machine that the sample from the FIFO is valid. The state machine will continue to load in the coefficient to the multiplier-accumulator until ORDY goes true, at which time the state machine will advance to the next state. If the cycle time of the multiplier-accumulator never exceeds the access time of the FIFO, ORDY should always be true when it's an input the state machine depends on.

Microcoded instructions to the multiplier-accumulator are generated by decoding the state variables. The first instruction is a NOOP. When SMPL_CONV goes low, then state machine issues a XBUS instruction to the multiplier-accumulator. This causes the multiplier-accumulator to load data from the I/O port into an internal register. The state machine then issues a YBUS;

CLKMR TC. This command tells the multiplier-accumulator to perform a multiply operation in twos-complement without accumulation because it's the first multiply operation of the convolution.

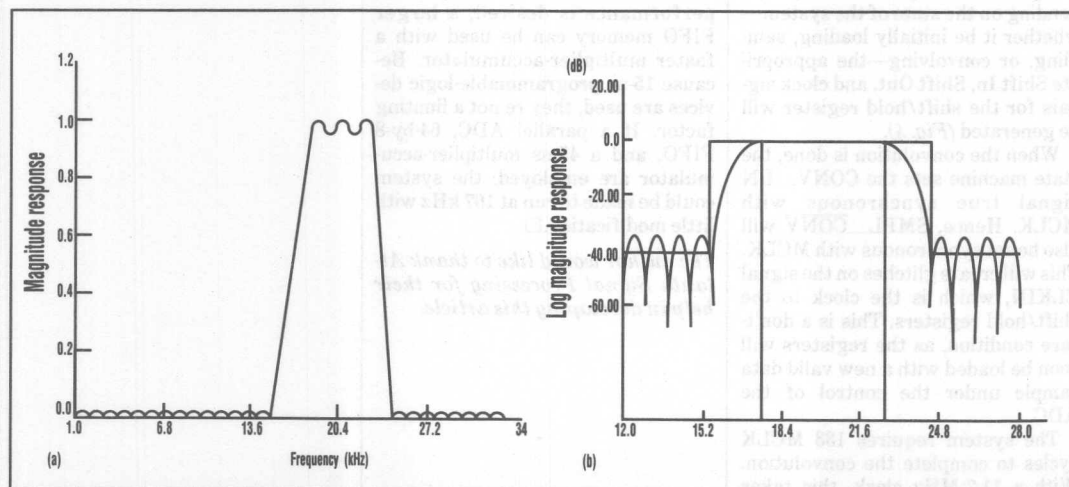
The machine then enters a loop and issues another XBUS command followed by a YBUS; CLMR; TC; MR+. This command is a multiply-accumulate function in twos-complement arithmetic. The machine remains in this loop until TC goes true, at which time the last multiplier-accumulator cycle is completed and the output command MS (SAT) is issued. MS causes the filter's outputs (multiplier-accumulator outputs) to become valid and latched into a final output register. This command will saturate the multiplier-accumulator output if the final value has an overflow, keeping the digital filter from oscillating. The multiplier-accumulator is statically configured to round off the final output to the most significant 16 bits.

The instructions to the multiplier-accumulator can be changed simply by decoding the state variables to different output values. If E²CMOS devices are used, the programmable device can simply be reprogrammed and put back into the circuit. An E²C-

MOS 22V10 from Lattice Semiconductor is one such device that can be used for this application.

Two 64-word-by-8-bit FIFOs can be used to implement the filter taps. The FIFO can be loaded up with the initial N samples. A sample is then shifted out of the FIFO and into the multiplier-accumulator for processing. This sample is also stored in a shift/hold register and is shifted back into the FIFO prior to the next sample being shifted into the multiplier-accumulator for processing. After all N samples have been processed, the oldest sample is shifted out and a new ADC sample shifted in. The multiplier-accumulator can then output a filter value. Programmable logic can be used to interface the digital filter to the ADC, act as temporary storage register, and implement FIFO control.

These shift/hold registers can be implemented with two 20V8 devices. In the sample mode (SMPL_CONV = 1), the devices act as shift registers. Data is serially loaded into them under control of the ADC. The registers are then placed in a hold mode so that the data sample isn't lost. When the system enters the convolve mode, (SMPL_CONV = 0), data is immediately loaded into the



5. A PLOT OF THE MAGNITUDE response shows that the bandpass filter's center frequency is 20 kHz with a passband of 5 kHz (a). The transition region occurred in 2 kHz. The log magnitude response plot reveals a 175-dB/decade slope at the edges of the filter (b). It would take a 9th-order analog filter to implement the same specifications.

DIGITAL FILTERS

shift/hold registers in parallel.

Filter coefficients are stored in PLDs emulating ROM. A 6001 has a programmable AND and a programmable OR array so that it easily emulates a 64-by-8 high-speed PROM. Again, if E² devices are used, the filter coefficients can be changed simply by reprogramming the devices. An address counter is used to access the coefficients in the correct order. Because there are 64 required coefficients for the 64 taps, only 6 bits of address are required.

The coefficient-address counter is a simple 6-bit counter implemented in a 22V10. The counter is a synchronous type with a count enable. The clock is synchronous with the multiplier-accumulator clock. The count-enable input pin is connected to XOE from the multiplier-accumulator controller. Therefore, the counter is incremented only after the coefficient value has been loaded into the multiplier-accumulator. When the counter reaches 63, TC goes true to indicate that all 64 coefficients have been convolved. Again, the power-up reset is used to ensure that the counter starts in a known state.

The remaining four output-logic macro cells can be used to generate FIFO control signals. These signals are generated asynchronously. Depending on the state of the system—whether it be initially loading, sampling, or convolving—the appropriate Shift In, Shift Out, and clock signals for the shift/hold register will be generated (Fig. 4).

When the convolution is done, the state machine sets the CONV_DN signal true synchronous with MCLK. Hence, SMPL_CONV will also be set synchronous with MCLK. This will create glitches on the signal CLKIN, which is the clock to the shift/hold registers. This is a don't-care condition, as the registers will soon be loaded with a new valid data sample under the control of the ADC.

The system requires 133 MCLK cycles to complete the convolution. With a 11.7-MHz clock, this takes 11.4 μ s. This system used an ADC with a serial interface that requires 3.3 μ s to shift the data into the shift/

hold registers. Thus, the system can sample an input signal at $11.4 + 3.3 = 14.7 \mu$ s or 68 kHz. The Nyquist sampling theorem states that a signal must be sampled at twice the highest frequency component to accurately preserve the information in that signal. Therefore, this system can accurately filter a signal with the frequency component as high as 34 kHz.

Using the DFD software from ASPL, a bandpass filter was designed using the Parks-McClellan method. The center frequency is at 20 kHz with a passband of 5 kHz. The transition region occurred in 2 kHz (Fig. 5). It's interesting to note that the edges of the filter have a slope of approximately 35 dB/0.2 decade, or 175 dB/decade. It would take a 9th-order analog filter to implement the same specifications.

The system presented in this example is a straightforward FIR filter. Because of the extensive use of programmable logic, the system can be easily adapted to implement an IIR filter. The final output value can be fed back into the FIFO prior to a new sample shifting into the FIFO. The coefficients can be staggered in the coefficient ROM so that the B_ms line up with the Y(n-M), and the A_ns line up with the X(n-N).

If enhancement of the system's performance is desired, a larger FIFO memory can be used with a faster multiplier-accumulator. Because 15-ns programmable-logic devices are used, they're not a limiting factor. If a parallel ADC, 64-by-8 FIFO, and a 45-ns multiplier-accumulator are employed, the system could be made to run at 167 kHz with little modification. □

The author would like to thank Atlanta Signal Processing for their help in developing this article.

SMPL_CONV, tell the machine when it's time to begin the convolution cycle. This signal comes from the mode-control signal TC (Terminal Count) which indicates when the convolution is done. TC comes from a 6-bit coefficient counter, and is valid when the count equals 63, which indicates when all 64 samples have been convolved with the respective coefficients. ORDY comes from the FIFO and tells the state machine that the sample from the FIFO is valid. The state machine will continue to load in the coefficient to the multiplier-accumulator until ORDY goes true, at which time the state machine will advance to the next state. If the cycle time of the multiplier-accumulator exceeds the access time of the FIFO, ORDY should always be true when it's an input the state machine depends on.

Microcoded instructions to the multiplier-accumulator are generated by decoding the state variables. The first instruction is a NOOP. When SMPL_CONV goes low, then state machine issues a XBUS instruction to the multiplier-accumulator. This causes the multiplier-accumulator to load data from the FIFO into an internal register. The state machine then issues a YBUS

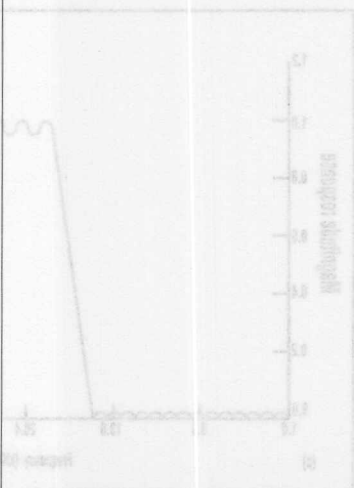


FIG. 5. A PLOT OF THE MAGNITUDE RESPONSE OF A FILTER. The magnitude response of a filter designed to implement the same specifications would take a 9th-order analog filter to implement the same specifications.

State Machine Design for High Speed PowerPC™ RISC Microprocessor Systems

Matt Carlson, Senior Design Engineer
Motorola

State Machine Design for High Speed PowerPC RISC Microprocessor Systems

This paper was presented at the 1994 PLD Design Conference and Exhibit in Santa Clara, California.

In order to pipeline addresses two levels deep, all first is required are a few latches for the holding the addresses and their associated attributes, and a pipeline controller. To design this pipeline controller to run at 75MHz, the state machine must meet a cycle time of 13.3 nanoseconds. This means that the clock-to-output delay (tCO) plus any setup time (tSU) to control registers must meet the 13.3 ns requirement.

$$tCO + tSU = 13.3 \text{ ns}$$

Finding a device that is complex enough to handle a large state machine and that can meet this critical speed requirement used to be a tough chore; however, there are now devices whose architectures are designed specifically for this purpose.

Designing for speed also requires using a device that can produce all output signals and state equations in a single level of logic. Using combinational feedback or latching logic out across several logic gate levels is too slow in standard PLDs. Another major decision involves deciding whether to use a FPGA or a CPLD.

The Demands of RISC
RISC microprocessors are demanding faster bus speeds than external logic can provide. In fact, the PowerPC 603™ and PowerPC 604™ processors are capable of running the internal clock of the part at two, three, and even four times the speed of the external bus, so that bus speed limitations do not slow down the internal speed of the part.

Bus speeds for this family are initially targeted at 80MHz, and should quickly evolve to 80MHz. The need for state machines to perform bus control functions at these speeds provides a challenge for systems designers.

In addition, the 603 & 604 external bus is capable of complex modes of operation; bus control designs can be built to use the most elegant and sophisticated features. Examples of some of these features are multi-processor arbitration, two- (or three-) level address pipeline control, split bus control, out-of-order bus transfer control, and transfer reply generation.

State Machine Design for High Speed PowerPCTM RISC Microprocessor Systems

Matt Carlson, Senior Design Engineer
Motorola

ABSTRACT

This presentation demonstrates design techniques for large, ultra-high speed (75 MHz) state machine designs used in RISC type bus architectures. Details of an application using CPLDs to implement large state machines for the PowerPC 603TM and PowerPC 604TM RISC microprocessor bus control will be described.

The Demands of RISC

RISC microprocessors are demanding faster bus speeds than external logic can provide. In fact, the PowerPC 603TM and PowerPC 604TM processors are capable of running the internal clock of the part at two, three, and even four times the speed of the external bus, so that bus speed limitations do not slow down the internal speed of the part.

Bus speeds for this family are initially targeted at 66MHz, and should quickly evolve to 80MHz. The need for state machines to perform bus control functions at these speeds provides a challenge for systems designers.

In addition, the 603 & 604 external bus is capable of complex modes of operation; bus control designs can be built to use the most elegant and sophisticated features. Examples of some of these features are multi-processor arbitration, two- (or three-) level address pipeline control, split bus control, out-of-order bus transfer control, and transfer reply generation.

To illustrate the techniques which show how to optimize a large state machine design, I will use the ABEL source file for a pipeline controller that has been developed for the PowerPC 603 and 604 microprocessors.

Pipelining addresses generated by the 603 or 604 processor greatly increases throughput. The costs of pipelining are minimal when compared with the relative increase in performance.

In order to pipeline addresses two levels deep, all that is required are a few latches for the holding the addresses and their associated attributes, and a pipeline controller. To design this pipeline controller to run at 75MHz, the state machine must meet a cycle time of 13.3 nanoseconds. This means that the clock-to-output delay (t_{CO}) plus any setup time (t_{SU}) to control registers must meet the 13.3 nsec requirement.

$$t_{CO} + t_{SU} = 13.3 \text{ nsec}$$

Finding a device that is complex enough to handle a large state machine and that can meet this critical speed requirement used to be a tough chore; however, there are a few devices whose architectures are designed specifically for this purpose.

Designing for speed also requires using a device that can produce all output signals and state equations in a single level of logic. Using combinatorial feedback or fanning logic out across several logic gate levels is too slow in standard PLDs. Another major decision involves deciding whether to use a FPGA or a CPLD.

Large State Machine- FPGA or CPLD?

FPGA devices typically run faster than CPLD devices if they can be used around a "one-hot" state bit encoding. However, if the state machine is complex (having many inputs or many state transitions), the FPGA with its "one-hot" encoding scheme falls short because the logic cell of the FPGA is too small and must resort to using multi-level logic delays to handle the number of inputs or product terms (pterms).

Because the speed requirement allows only one level of logic, a design which has a large number of pterms cannot be implemented well in an FPGA. This is the case with the pipeline controller for the 603/604. It has 34 states, 93 state transitions, 13 inputs, and 12 outputs. This is more than twice as large as the "Large State Machine" of PREP's benchmark #1.

A CPLD is a better choice for large state machines than FPGA architectures for yet another reason. Because the CPLD provides a higher number of pterms in each cell than an FPGA, a maximal (or near-maximal) state bit encoding fits more easily. The low number of state bits in the CPLD can very likely all be brought out on I/O pins for external visibility. An FPGA with a "one-hot" encoding is probably not able to have all its internal state bits visible on the I/O pins.

Another very important point in using CPLDs for fast machines is that they boast a very predictable delay path. You know that the design will meet speed requirements before you even start to implement it in the device. This is a big advantage over an FPGA because FPGA architectures typically depend on the routing delays to provide final results of the speed of operation. These FPGA routing delays aren't known until the design has been completely implemented in the device— not a good time to find out that there will be much more routing work to do!

Design Definition

The ABEL source file format for the pipe controller is shown in Figure 1.

```
Module PIPECNTL
Title 'PIPECNTL— TC60X Pipeline Controller
Matt Carlson & Gregg Mack Copyright Motorola 1993'

"INPUTS
clk, reset, tts, ts, aack, artry, sdead, srw, sdone,
anydbr, dbwo, mrw, nrw PIN;

"OUTPUTS
sts, ok2kill, ok2ta, noe, holdn, holds, bpoe,
do_dbwo, pipe0, pipe1, pipe2, pipefull PIN;

"STATE BIT REGISTERS
sb0, sb1, sb2, sb3, sb4, sb5, sb6 NODE;

"STATE ASSIGNMENTS
st0 = (!sb0 & !sb1 & !sb2 & !sb3 & !sb4 & !sb5 & !sb6);
st1 = (!sb0 & !sb1 & sb2 & !sb3 & !sb4 & !sb5 & sb6);
st2 = (!sb0 & sb1 & !sb3 & sb4 & !sb5);
st3 = (sb0 & sb1 & !sb3 & sb4 & !sb5);
:
st33 = (!sb0 & !sb1 & sb2 & sb3 & !sb4 & sb5);

Equations
" st1 := 7 Pterms
"
" st0 & ts
" # st1 & !aack & !anydbr
" # st4 & ts & sdone
" # st5 & !aack & !anydbr & sdone
" # st10 & ts & sdead
" # st13 & ts & sdead
" # st15 & sdead ;

sb2 :=
" st1 := 7 Pterms
"
" st0 & ts
" # st1 & !aack & !anydbr
" # st4 & ts & sdone
" # st5 & !aack & !anydbr & sdone
" # st10 & ts & sdead
" # st13 & ts & sdead
" # st15 & sdead

" st6 := 2 Pterms
" # st5 & !aack & anydbr & sdone
" # st19 & !aack & anydbr & sdead

" st7 := 1 Pterms
" # st6 & !aack

" st33 := 2 Pterms
" # st32
" # st33 & !sdone ;

:
End PIPECNTL
```

Figure 1: ABEL Source file format for Pipe Controller

The inputs and outputs are defined simply as "PINS" and the state bits are openly defined as "NODES". A state assignment table defines each state (stx) as a function of the state bits (sbx); this table serves to let state assignments be made and rearranged easily.

The state equations follow the state assignment table. These state equations are simply input straight from the state diagram—one pterm for each arrow on the state diagram. The equations are shown in Boolean format (no HDL constructs) so that simplification of the equations can be easily seen. The state equation format is:

```
stW := stX & A & B & ...
      # stY & C & D & ...
      # stZ & E & F & ... ;
```

where stW is the next state; stX, stY, and stZ are previous state; and A-F... are inputs. The state bit equations (sbx) follow the state equations and the equations for the output signals are listed last.

Preliminary Simplification

The first step in reducing the equations is the same step as performed by the Quine-McCluskey method—eliminate literals using the reduction theorem:

$$XY + XY' = X \quad [2]$$

For example, Figure 2 shows how this is done with the pipe controller output equation HOLDS. Pterms which are a function of the same state are compared; the two pterms with st4 (state 4) are combined, the three pterms with st5 are combined, and so forth. Simplifying the output equation in this manner reduces the number of pterms from 45 to 31.

State Pairing Technique

The second step is to look at the pterms in each equation and group pterms that have identical inputs but have different states. This pairing technique aims at setting up the state bit assignment so that the second part of the Quine-McCluskey method will

achieve the most pterm reduction. The simplified HOLDS equation has the first two pterms with identical inputs yet different states:

```
HOLDS := st2 & aack
        # st3 & aack
        # ...
```

State 2 and state 3 are "paired" together with the intent of assigning their state bits such that they only differ in one bit.

HOLDS := " 45 -> 31 Pterms

```
# st2 & aack
# st3 & aack
# st4 & !ts & !sdone
# st4 & ts & !sdone
# st5 & aack & !anydbr & !sdone
# st5 & aack & anydbr & !sdone
# st5 & !aack & !sdone
# st7 & aack & !sdead
# st8 & artry & !sdone
# st8 & !ts & !artry & !sdone
# st8 & ts & !sdone
# st9 & !ts
# st10 & ts
# st10 & !ts & !sdead
# st10 & ts & !sdead
# st11 & !ts
# st11 & ts
# st12 & !artry
# st13 & !ts & !sdead
# st13 & ts & !sdead
# st14
# st15 & !sdead
# st16 & aack & !anydbr
# st16 & aack & anydbr
# st16 & !aack
# st17 & !sdone & !dbwom
# st17 & !sdone & mrw
# st17 & !sdone & !nrw
# st17 & sdone & dbwom & !mrw & nrw
# st17 & !sdone & dbwom & !mrw & nrw
# st18
# st19 & aack & anydbr & !sdead
# st19 & !aack & anydbr & !sdead
# st19 & !anydbr & !sdead
# st20 & artry
# st20 & !artry
# st21 & ts & !sdone
# st21 & !ts & !sdone
# st23 & !sdone
# st24 & !sdone
# st25
# st26 & !aack
# st26 & aack & anydbr & !sdead
# st28
# st29 ;
```

Figure 2: Combining Product Terms of initial HOLDS using the Reduction Theorem.

This pairing will allow these two pterms to be combined after state bit assignment. If there are three or more pterms that match, they are grouped together and assigned an encoding so that as many state bits as possible differ in only one bit.

For example, the four pterms st10, st13, st15, and st19 in the simplified HOLDS equation (Figure 2) can be grouped together since they have identical inputs but different previous states. They should have a state bit assignment of the form:

```
st10 = xxxxx00
st13 = xxxxx01
st15 = xxxxx11
st19 = xxxxx10
```

These four pterms could then be reduced to a single pterm which would be:

```
xxxxx & !sdead
```

This pterm grouping procedure should be done for all state equations and output equations.

How Many State Bits to Use?

After grouping the states as described above, the third step is to determine how many state bits to use for a maximal (or near maximal) encoding. The number of state bits in a maximal encoding is:

$$N = \text{RND}(\log_2 S),$$

where S is the total number of states and RND rounds the number up to the nearest integer. For the pipe controller, 34 states could be maximally encoded by 6 state bits.

However, a maximal encoding is usually not a good choice when the number of pterms is limited by the architecture. The number of pterms in the state bit equations will increase drastically if there are many "1"s in their state bit assignments. Keeping the number of ones in the state bit assignments to a minimum is very important to understand. In fact,

this idea is carried to the extreme in a "one-hot" state bit assignment where there is only one active "1" in any state.

The best choice for the number of state bits in the CPLD is "near" maximal—usually N+1 or N+2. This provides a good selection of state bit combinations that have a low occurrence of "1"s. Accordingly, a low number of state bits also allows all of the state bits to be brought out to I/O pins for external real-time visibility (as opposed to having them "hidden" in buried registers). We use N+1 (seven) for the pipe controller which allows us to map 34 states into 128 possible state bit encodings.

State Bit Assignment Method

The fourth step is to make the actual state bit assignment. This is done in a table; each state is listed along the left column and the state bits are listed along the top. Figure 3 shows a portion of the state assignment table for the pipe controller. The states (st0 through st33) are along the left and the state bits (SB0 through SB6) are along the top.

Also along the left column is the estimated number of pterms for each state. Each estimated number is conservatively determined as the original (unreduced) number of pterms minus the number of state pairs associated with that state.

For example, in the equation for st1 of Figure 1, the original number of pterms was seven. The pairing of st10 with st13 will reduce the equation to six pterms, so the estimated number of pterms for st1 in the table is six.

At the bottom of the state assignment table is the total number of pterms for each state bit. Wherever a "1" occurs in the assignment table, the corresponding number of estimated pterms for that state must be added to the total at the bottom of the table. This total is therefore an estimated total. Note that "0"s and "x"s do not add to the estimated total.

As the state bit assignments are made, there are three main things to consider:

- 1) Give higher priority to pterm pairs which occur multiple times over those that occur only once.
- 2) Assign few "1"s in states that have a high number of estimated pterms.
- 3) Keep a running total of the estimated number of pterms for each state bit column and try to keep that number at or below the maximum number allowed by the targeted CPLD architecture.

		S	S	S	S	S	S	S
		B	B	B	B	B	B	B
pterm	state	0	1	2	3	4	5	6
13	st0	0	0	0	0	0	0	0
6	st1	0	0	1	0	0	0	1
2	st2	0	1	x	0	1	0	x
1	st3	1	1	x	0	1	0	x
8	st4	0	1	x	0	0	0	x
4	st5	0	1	x	0	0	1	x
2	st6	0	0	1	0	1	0	0
1	st7	0	0	1	0	0	1	1
4	st8	1	0	x	0	0	0	0
3	st9	0	0	0	0	0	1	x
1	st10	0	0	0	1	0	1	x
2	st11	0	0	0	1	1	0	x
.
.
2	st33	0	0	1	1	0	1	x
Est. Total:		30	28	29	20	21	30	28
Actual Total:		16	19	17	18	17	19	19

Figure 3: State Bit Assignment Table

Product Term Results

Before the actual number of pterms can be determined, the actual state bit equations must be entered into the ABEL source file directly from the state bit assignment table in the form (see Figure 1):

```
SBx := {stx equation}
# {sty equation}
# {stz equation}
# ... ;
```

where "SBx" is a state bit, and "{stx equation}" is the entire state equation wherever a "1" appears in the SBx column of the state assignment table in Figure 3. For example, the equation for SB2 is:

```
SB2 := {st1 equation}
# {st6 equation}
# {st7 equation}
:
# {st33 equation} ;
```

Expanding each state equation {stx equation}, this becomes:

```
SB2 :=
{
# st0 & ts
# st1 & laack & lanydbr
# st4 & ts & sdone
# st5 & laack & lanydbr & sdone
# st10 & ts & sdead
# st13 & ts & sdead
# st15 & sdead } "(st1)
{ # st5 & laack & anydbr & sdone
# st19 & laack & anydbr & sdead } "(st6)
{ # st6 & laack
# st5 & aack & anydbr & sdone
# st6 & aack
# st19 & aack & anydbr & sdead } "(st12)
:
{ # st33 & lsdone } "(st33)
;
```

This expansion is done for each state bit equation and entered into the ABEL source file. ABEL automatically substitutes the state bits (SB0...SB6) for each state (st0...st33) from the assignments in the state table.

Below the estimated totals at the bottom of the state assignment table is the actual number of pterms. The actual number of pterms is determined by entering the state bit assignments into the state table of the ABEL source file and then running ABEL Espresso for the result. ABEL has a utility called "pla2eqn" that will generate a minimized solution in Boolean equation form along with a table which shows the number of pterms for each equation.

Once a first pass reduction of the equations is complete, a choice of architectures should be made in which to implement the state machine design.

Architecture Selection

The main goal of design implementation is to make it fit into one or more devices and run at the required speed. The requirements for the pipe controller were to find a device with at least 20 pterms per register that could run at 75 MHz.

Two devices available at the time which could meet these criteria were AMD's MACH 210 and the Lattice pLSI 1016. The MACH device provides 16 pterms to each macrocell, whereas Lattice's pLSI device provides 20 pterms to each GLB (generic logic block). The additional 4 pterms per logic block offered by the Lattice pLSI device is a major advantage over the MACH 210. Routability, number of I/O pins, reprogrammability, and number of logic blocks also favored the pLSI.

The Final Fit

Once a device is chosen, the design can be fine-tuned to fit its architecture. If the number of pterms is too large on one or more of the state bit equations, a couple of options exist which will allow the design to fit:

- 1) Modify state bit assignments to evenly distribute pterms and run the modified state assignment back through ABEL again.
- 2) If a few iterations of step 1 prove fruitless, add another state bit to lower the number of pterms in each state bit.

After a solution to the number of pterms per state bit equation has been identified, the number of inputs to each equation must not exceed the number allowed by the architecture. The Lattice pLSI is generous with 18 inputs to each logic cell.

Implementation

Once the state bit assignment has been finalized and the number of pterms and inputs to each equation fits the architecture, the output from ABEL is used for input to the Lattice pDS routing software. The ".eqn" output file from

ABEL is simply "cut and pasted" into global logic blocks (GLBs) of the pLSI device. Pin I/Os can either be locked into place by the user or left to be assigned by the routing software. The best routing occurs if the pins are assigned by the router.

This particular design routed in approximately two hours running on a 486SX-33MHz PC. As soon as the route was complete, it was comforting to know at that point that the design was finished. There was no question that the pipe controller would run at least 75 MHz.

Summary

High speed state machines can provide the bus control needed for optimal PowerPC performance. The Lattice pLSI 1016 is an excellent choice for implementing these high speed controllers. The state bit assignment and logic reduction techniques provide an excellent and practical method of compacting a large state machine into the CPLD architecture.

References:

- [1] Benchmark Suite #1, Revision 1.2, Programmable Electronics Performance Corporation, 1993.
- [2] Charles H. Roth, Fundamentals of Logic Design, 2nd ed., 1983, pp. 55-57.
- [3] "Lattice Data Book." 1994. Lattice Semiconductor Corporation, Hillsboro, Oregon.
- [4] PowerPC Architecture Definition, 603 Book IV, revision 3.3.
- [5] PowerPC Architecture Definition, 604 Book IV, revision 3.1.

In this article, the terms "PowerPC 603 Microprocessor" and "603" and the terms "PowerPC 604 Microprocessor" and "604" are used to denote the second and third implementations, respectively, of the PowerPC architecture. PowerPC, PowerPC 603 and PowerPC 604 are trademarks of IBM Corp. and are used by Motorola under license from IBM Corp.

paper was presented at the 1994 PLD Design Conference.

Applying In-System Reprogrammability in a REFLECTIVE MEMORY™ Bus Controller

Steve Schelong
Hardware Engineer
Encore Computer Company

Introduction

Today's complex system designs are ever more demanding of speed, integration levels, and testability. The less technical issues of time-to-market and development costs are equally important. These constraints provide the driving force behind the boom in use of programmable logic devices. Encore Computer's "Infinity 90™" is an example of a complex computing system which has benefited from the incorporation of programmable logic devices.

The Infinity System is a multi-computing system developed for use in computational intensive applications such as flight simulation and real-time data bases. It handles such tasks as graphics generation, motion control, real-time data acquisition/processing, and concurrent search algorithms. This high-

performance parallel computer was designed to solve the shortcomings of the mainframe computers currently in use for these tasks.

Encore's computer is comprised of a collection of computing elements which communicate over a shared REFLECTIVE MEMORY (RM) bus (Figure 1). Each computing element is an autonomous computer with internal busses and local memory. Hierarchies of computing regions can be established with the use of FRC bridge units and fiber-optic or coax links. Although each computing element is autonomous in its operation, data coherency must be maintained amongst the elements. This coherency is enforced through the use of the global writes over the RM bus.

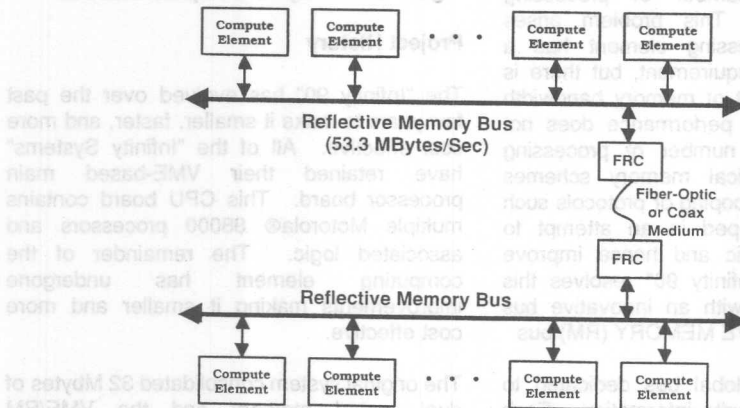


Figure 1. Encore's Infinity 90 System Incorporating REFLECTIVE MEMORY Bus

Implementation of this system makes effective use of in-system programmable (ISP™) logic devices as controlling logic within the computing element. The use of high density PLDs creates a streamlined system which minimizes board space and operates at high speed. Time to market issues are addressed by short development cycles and easy bug fixes. In-system programmability allows Encore to provide on-site customer enhancements and updates without replacing or modifying hardware. A detailed analysis of design problems, alternatives and final solutions are presented in the following sections.

[What is] REFLECTIVE MEMORY

The "Infinity 90" parallel computer requires a method for transporting and distributing data amongst computing elements. Traditional methods of providing this communication are divided into two categories: shared-memory systems and message passing systems. The "Infinity 90" falls into the former category. Shared-memory systems, however, often suffer from memory bandwidth limitations, especially if the number of processing elements is high. This problem arises because each processing element has a memory bandwidth requirement, but there is only a limited amount of memory bandwidth available. Typically, performance does not scale well with the number of processing elements. Hierarchical memory schemes employing caching/snooping or protocols such as SCI were developed in an attempt to minimize global traffic and hence improve performance. The "Infinity 90" resolves this and other problems with an innovative bus called the REFLECTIVE MEMORY (RM) bus.

The RM bus is a global bus dedicated to distributing memory write information. Each computing element (figure 2) contains a local VME bus which operates at VME speeds. Write information on the VME bus is queued within a RM bus interface. This interface bids for use of the RM bus through global arbitration. When the RM bus is granted, the interface broadcasts the write information to all other computing elements which then updates their local memories. In this manner,

all writes are reflected to all computing elements and memory coherency is maintained. It allows the computing elements to operate at local bus speeds, unrestricted by global bus penalties.

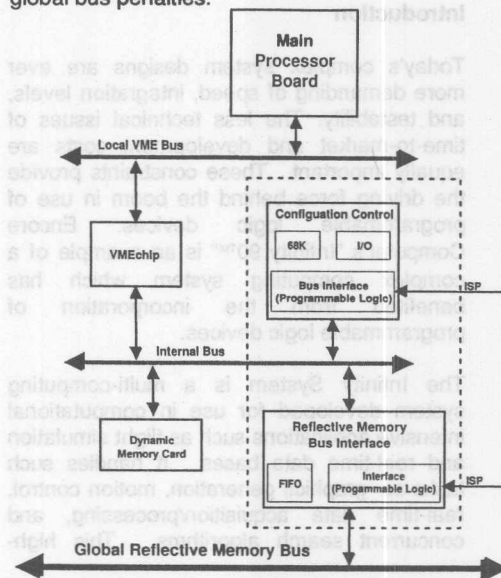


Figure 2. Infinity 90 Compute Element

Project History

The "Infinity 90" has evolved over the past few years to make it smaller, faster, and more cost effective. All of the "Infinity Systems" have retained their VME-based main processor board. This CPU board contains multiple Motorola® 88000 processors and associated logic. The remainder of the computing element has undergone improvements making it smaller and more cost effective.

The original system consolidated 32 Mbytes of dual ported memory and the VME/RM interface on a VME 9U size board. This implementation required ten 22V10s, twelve 20-pin PLDs, and numerous 74xx series components. These components alone required 10 square inches of board space. Power consumption was understandably high. Although higher density components were available at the time, these components were not capable of operating at the required

speeds. To reduce physical size, power consumption, and improve performance, development of a new implementation of the compute element was undertaken. This implementation, wherever possible, was to leverage off components and boards which were available as off-the-shelf items. This decision reduced development costs and improved time-to-market. The entire system (not including the main processor board) was targeted to fit onto a VME 6U card. Figure 2 shows the internal architecture of the new computing element.

To take advantage of pre-designed subsystems, the Motorola Mezzanine daughter card was chosen as the dynamic memory subsystem. This card allowed user to choose from 4 to 512 Mbytes of memory depending on system requirements. Rather than develop a customized VME bus interface, the Motorola VMEchip2 was chosen. This decision has both good points and bad points. Although the VMEchip2 provides a full VME Rev D implementation in a highly integrated package, it also requires a dedicated subsystem for initializing and configuring the VMEchip at startup. This configuration control block is shown in detail in figure 3.

The configuration control block is designed using a 68000 processor with local memory, an asynchronous I/O port, and a custom bus-interface to adapt the 68000 bus to the computing element's internal bus. The bus-interface logic incorporates Lattice's high density PLD components for address decode, interrupt, handshake and request logic functions (figure 4).

The remainder of the system is composed of a custom implementation of the RM bus interface. Its architecture is shown in figure 5. This interface primarily consists of FIFOs to queue data transactions, RAMs for mapping, and control logic for sequencing and interfacing. The control logic portion utilizes high density PLDs. Although the architecture of this RM interface is essentially the same as that of the previous generation compute elements, the level of integration is much higher and overall flexibility is improved by reprogrammability. This improvement is

derived from the benefits of fast, high density logic and in-system reprogrammability. The tasks assigned to the programmable logic devices are shown in figure 6. The blocks within the dashed box of figure 2 are the subsystems of the compute element which have benefited from use of today's high density programmable logic. Experiences and examples of their implementation are outlined in the following sections.

Project Constraints

Redesigning of the compute element involved physical and time constraints that forced the investigation of new ways to implement logic which was previously accomplished using low density PLDs and SSI/MSI components. As previously mentioned, the first version of the compute element used 22 low density PLDs and an assortment of 74xx devices. This required 10 square inches of board space. Moving the system from a 9U board to a 6U board would not allow for 10 square inches of board space just for logic. A higher density solution was needed.

Although design solutions using low density devices were sufficiently fast, their use had been ruled out for area and power reasons. This created concern as to whether it would be possible to create a system fast enough using higher density programmable logic solutions. The logic would need to provide substantial decode and sequencing functions for bus speeds of 33 MHz. Gate-array solutions would certainly be fast enough, but the project had neither the budget to absorb high NRE costs or the time for long development cycles. The previous version of the system demonstrated that the speed of high density programmable logic [available at the time] was insufficient. Fortunately, since then, the speed of certain types of these devices has improved considerably.

Using Motorola's Mezzanine memory system and VMEchip2 from the 187 Processor board saved the project significant design resources and time. These boards come out of Motorola's board division and not out of a chip division. Unfortunately, application support was not available for these subsystems and the documentation was not always clear. This

created uncertainty in the subtleties of their operation. An initial project goal was to simulate the entire system, but without a better understanding of component operation, simulation was not possible. The only reasonable alternative was to design a prototype using the assumed operation and fix things along the way. This implied that significant logic changes were likely along with multiple design iterations.

As is the case with most projects, there is never enough time allotted. The redesign of the compute element is no exception. The project was given seven months to take the redesign from specification to final implementation which was a very tight schedule.

Technology Choices

In the beginning of the redesign, Encore Computer determined that the technology choice for the new implementation would be of higher density than the low density PLD solution of the first version. In addition to area and power considerations was the concern that design changes were likely to occur late in the design cycle. Using low density parts, logic changes would likely result in PC board layout changes causing schedule slips. High density devices were desirable since they could integrate much larger portions of the control circuitry. This would reduce the chance of having to go outside the device if changes were needed.

The first alternate to be investigated was FPGA technology. This technology offered some very desirable attributes, the greatest of which was the level of integration attainable with the device's high gate count. They also allowed for changes and bug fixes by simply reprogramming (or replacing of a chip). Design proceeded using FPGAs but portions of the design showed significant signal delays when passing through the FPGA. This caused immediate concern since operation at 33 MHz required very fast signals. As development continued, signal delays surfaced as another issue. Signal delays, even if logic had been optimized to meet timing constraints, changed from one routing iteration to another. Portions of the circuitry deemed complete couldn't be

considered finished since timing would change as new circuits were added to the FPGA. Timing was uncertain until the very final route. In addition, routes were taking as long as 24 hours and the project was under considerable time constraints. The combination of these difficulties prompted the project team to begin looking for alternatives to FPGAs.

High density PLDs are now much faster and denser since those investigated in the first version of the compute element. Several device vendors had products which seemed suitable for this application. After evaluating the choices, Encore Computer decided to use the 1000 family of high density PLDs from Lattice Semiconductor. The deciding factors were device speed and use of in-system programmability (ISP). The programmability and reprogrammability feature would be beneficial since several design iterations were expected. Since ISP devices can be soldered directly onto the PCB, changes could be affected without removing the devices. This eliminated the need for expensive, unreliable sockets and provided confidence that if something wasn't working it wasn't due to bad socketing. It also reduced the chance of disturbing the test setup from the mechanical action of removing and replacing chips.

As the design proceeded it was soon discovered that portions of the logic design of the previous version could be directly re-used in this design. The previous design was implemented using PLDs programmed with Data I/O's ABEL system. This ABEL software could also be used to program the Lattice high density devices. Since the structure of these devices is a superset of PLDs, much of the ABEL code mapped directly into the high density devices. This reduced the task of re-mapping the logic into another structure. Control logic operation speed was a constant concern and was a driving force which motivated the use of high speed PLDs in the first version of the system. The current high density devices promised speeds approaching those of the fastest low density PLDs. Yet they provided higher levels of integration. The chosen Lattice parts were fast enough to implement all the control functions without having to implement critical functions in specialized external logic. All sections of

control logic could be logically partitioned and mapped into the devices in a structured fashion. These circuits operated to design specifications.

A problem encountered with FPGAs was their non-deterministic delays encountered between routing iterations. It was frustrating to find that after waiting 24 hours for a route to complete, delays which had been acceptable had now increased to be unacceptable. This problem was not experienced with the high density PLDs. Not only did the high density PLDs route in less than one hour, but once the delays were established for a logic section, those delays would remain fixed between routing iterations. It was possible to rely on timing information remaining stable as the design proceeded.

During implementation, the project schedule expanded to nine months and the project was completed in that time. Design of the system and control logic were successfully concluded without major incident. Timing, power, and area constraints were met and the system was ready to ship.

ISP helped the project during the design and implementation stage. Once the project was complete, a secondary benefit of ISP was realized. Systems had shipped to customers, but enhancements and upgrades in the hardware are ongoing. The installation of these upgrades had previously required that PLDs be pulled from the boards and be replaced with updated PLDs - a labor intensive task involving system disassembly. With ISP, it is now possible to provide these updates by simply mailing a disk containing the updated JEDEC file to the customer site. The customer then downloads the JEDEC file from a personal computer and the change is affected. Thus streamlining the process of providing field updates and reducing associated costs.

Conclusion

Encore Computer's parallel computing system, "Infinity 90", is now in its second generation. The first generation relied on the use of low density PLDs, while the second generation machine was streamlined to

improve performance and reduce costs. To realize these improvements, Encore Computer effectively used in-system programmable high density PLDs. These devices are used as controlling logic in the RM bus interface and system configuration subsystems. The requirement for high speed operation limited the choice of programmable logic technologies. Lattice Semiconductor's ispLSI 1000 devices provided both the speed and density required for these applications. Logic design was simplified because of uniform and predictable delays through these devices. In-system programmability provided dual benefits. It resulted in fast debug and update cycles as well as providing an easy path for field upgrades. The design of the system proceeded on schedule and resulted in fast time-to-market.

ISP is a trademark of Lattice Semiconductor Corporation.

REFLECTIVE MEMORY and Infinity 90 are a trademark of Encore Computer.

Motorola is a registered trademark of Motorola Incorporated.

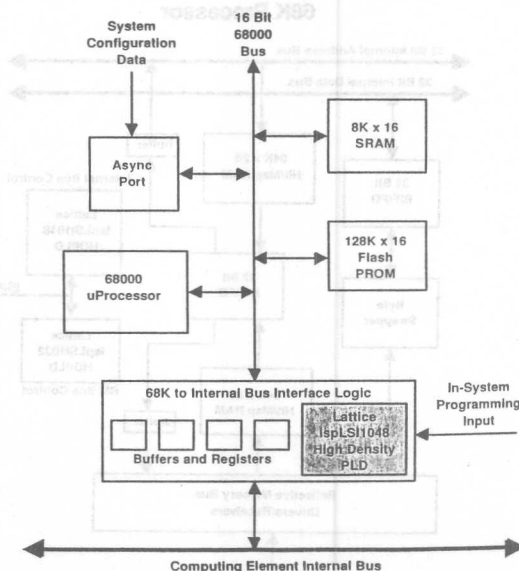


Figure 3. Initialization and Configuration Subsystem

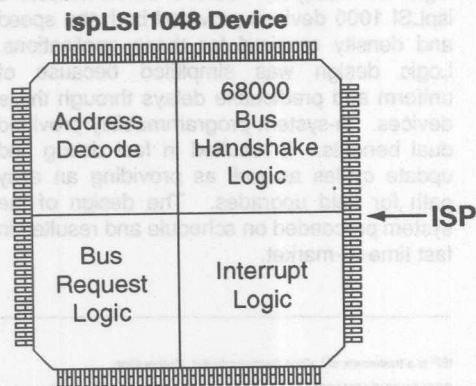


Figure 4. Logic Functions within Lattice ispLSI 1048 for Internal Bus Interface to 68K Processor

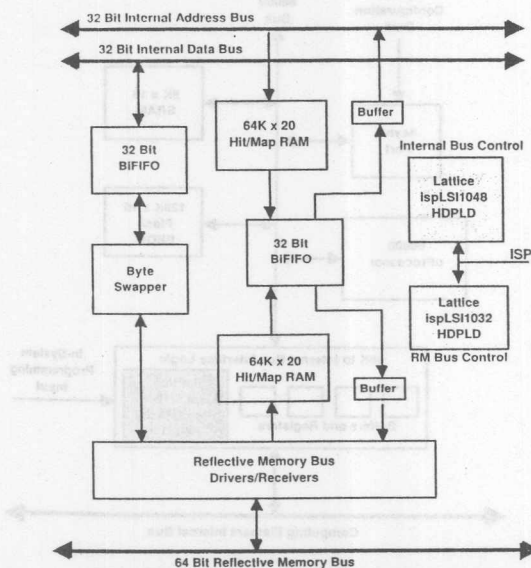


Figure 5. REFLECTIVE MEMORY Bus Interface

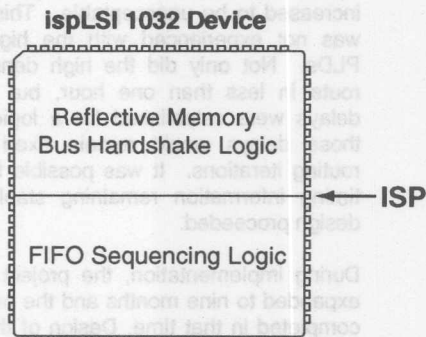


Figure 6a. Logic Functions within Lattice ispLSI 1032 for RM Bus

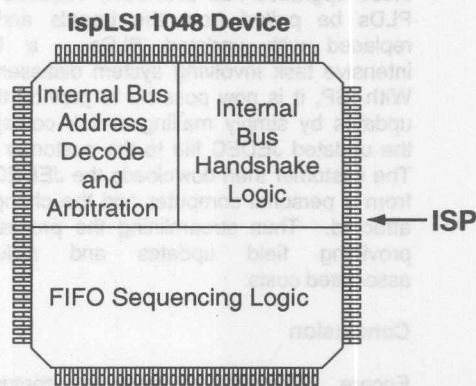


Figure 6b. Logic Functions within Lattice ispLSI 1048 for RM Bus

PLD Usage Generalizes HDTV Frame Buffer Interface

David L. Hagner - Senior Design Engineer
Convex Computer Corporation

Design Goals

The Special Systems group of Convex, whose charter is to develop custom interfaces to satisfy customer requirements, already had some experience with frame buffer

interfaces to the image Sequencer and Storage Processor (ISP), manufactured by VLSI GmbH. That design required TTL data paths that could handle up to 40 Mbytes per second. The design was conceived for the development of a custom interface to the VLSI Silicon Front-End (VSE) which is manufactured by VTE GmbH. Although both the ISP and VSE offer similar features to the end user, the design of the custom interface to the VSE requires a 16 bit differential ECL data and control path.

From the start of the project two goals were identified as being key to the success of the design. The first was to produce a single interface that could handle both the VSE and VTE frame buffers. The second was to double the transfer rate from the 40 Mbytes per second on the original VSE interface to 80 Mbytes per second. This would allow Special Systems to offer a single solution to a customer who could then select which frame buffer best matched the current research thrust. It would also give a higher performance upgrade path for existing customers.

Convex IO Structure

The Convex IO system is based on a series of Channel Control Units, which supply a 64 bit, 80 Mbytes per second synchronous data path to an attached peripheral or subsystem. In most cases, custom interface designs use a High Speed Parallel (HSP) channel controller and an additional channel controller attached to the CCU through a 100 foot cable (Figure 1). The channel, known as the HSP Interface Adapter (HIA), mounts in a standard 19" rack and provides a convenient platform for interface boards designed on a single high Eurocard form factor. Four slots of the nine slot card cage are dedicated to a standard board set. This board set provides the designer a choice of interface options plus 64 Kbytes of FIFO buffering which eases the design task when differing data rates are involved.

Approximately two years ago, the Special Systems group undertook a program to shorten custom design cycles and reduce costs. This effort resulted in a two point set which could be used in a majority of the applications. The first board, called the LCA Universal Interface (LUI), is based on four Xilinx XC3064-100 FPGAs and extends

Background

High Definition Television (HDTV) is an emerging technology that is being eagerly pursued by many companies around the world. The very name invokes images of

However, beyond the computer electronics arena, there is a vast area of opportunity in both business and defense applications. For example, the possibility of using HDTV from space, and at least one defense application, the technology for

Currently, many of the needs for HDTV have been defined, such as aspect ratio and resolution. However, no quantity many companies are willing to gamble millions of research dollars now, in hopes of sharing a piece of what promises to be a multibillion dollar industry by the early 21st century. Although many preliminary systems exist in the lab and several have even advanced far enough for field trials, none of the competing approaches have established a dominating position. Many of the systems offer significant advantages, such as compatibility with existing standards or a host of new and exciting features. Unfortunately, all of the systems face similar problems, such as dealing with a medium that requires 16 times more pixels than conventional television and bandwidths that are on the order of 360 Mbytes per second. Therefore, current research focuses on areas such as the development of error correction and data compression algorithms. This research requires a tremendous amount of computing power.

The computers used by HDTV research organizations have two primary purposes. First they are used as simulation engines to determine how a given system will react to real world problems, such as noisy transmission medium or component failure. Second they are used to transmit digital video images to a display device that is connected to an IO channel. These transmission requirements strain the IO capabilities of today's computers and this has given rise to a peripheral that has been instrumental in the development of HDTV. This peripheral is called a Frame Buffer and is a very large, fast data store with some specialized processing capability. Frame buffers which meet the needs of HDTV research are currently available from several manufacturers. This paper will address the development of an interface between a Convex IO channel and two of the more popular frame buffers.

PLD Usage Generalizes HDTV Frame Buffer Interface

David L. Harper - Senior Design Engineer
Convex Computer Corporation

Background

High Definition Television (HDTV) is an emerging technology that is being eagerly pursued by many companies around the world. The very name invokes images of theater quality video and sound brought into the home. However, beyond the consumer electronics arena, there is a vast area of opportunity in both business and defense applications. For example, NASA is looking at the possibility of using HDTV for pictures transmitted back from space, and at least one European firm is looking at the technology for video phone and videoconferencing applications.

Currently, many of the goals for HDTV have been defined, such as aspect ratio and resolution, however, no standards yet exist regarding implementation. Consequently, many companies are willing to gamble millions of research dollars now, in hopes of sharing a piece of what promises to be a multibillion dollar industry by the early 21st century. Although many preliminary systems exist in the lab and several have even advanced far enough for field trials, none of the competing approaches have established a dominating position. Many of the systems offer significant advantages, such as compatibility with existing standards or a host of new and exciting features. Unfortunately, all of the systems face similar problems, such as dealing with a medium that requires 15 times more pixels than conventional television and bandwidths that are on the order of 200 MBytes per second. Therefore, current research focuses on areas such as the development of error correction and data compression algorithms. This research requires a tremendous amount of computing power.

The computers used by HDTV research organizations have two primary purposes. First, they are used as simulation engines to determine how a given system will react to real world problems, such as a noisy transmission medium or component failure. Second, they are used to transmit digital video images to a display device that is connected to an I/O channel. These transmission requirements strain the I/O capabilities of today's computers and this has given rise to a peripheral that has been instrumental in the development of HDTV. This peripheral is called a Frame Buffer and is a very large, fast data store with some specialized processing capability. Frame buffers which meet the needs of HDTV research are currently available from several manufacturers. This paper will address the development of an interface between a Convex I/O channel and two of the more popular frame buffers.

Design Goals

The Special Systems group at Convex, whose charter is to develop custom interfaces to satisfy customer requirements, already had some experience with frame buffer interface design. Two years ago the group developed an interface to the Image Sequence and Storage Processor (ISP), manufactured by DVS GmbH. That design required a 64 bit, single-ended TTL data path that could transfer bi-directional data at speeds up to 40 MBytes per second. Recently, an order was received for the development of an interface to the Digital Video Silicon Recorder (DVSR), which is manufactured by VTE GmbH. Although both the ISP and DVSR offer similar features to the end user, the interface is completely different; the DVSR requires a 16 bit differential ECL data and control path.

From the start of the project two goals were identified as being key to the success of the design. The first was to produce a single interface that could handle both the DVS and VTE frame buffers. The second was to double the transfer rate from the 40 MBytes per second on the original DVS interface to 80 MBytes per second. This would allow Special Systems to offer a single solution to a customer who could then select which frame buffer best matched his current research thrust. It would also give a higher performance upgrade path for existing customers.

Convex I/O Structure

The Convex I/O system is based on a series of Channel Control Units, which supply a 64 bit, 80 MByte per second synchronous data path to an attached peripheral or subsystem. In most cases, custom interface designs use a High Speed Parallel (HSP) channel controller and an additional chassis remotely attached to this CCU through a 100 foot cable set (Figure 1). The chassis, known as the HSP Interface Adapter (HIA), mounts in a standard 19" rack and provides a convenient platform for interface boards designed on a triple high Eurocard form factor. Four slots of the nine slot card cage are dedicated to a standard board set. This board set provides the designer a choice of interface options plus 64KBytes of FIFO buffering which eases the design task when differing data rates are involved.

Approximately two years ago, the Special Systems group undertook a program to shorten custom design cycles and reduce costs. This effort resulted in a two board set which could be used in a majority of the applications. The first board, called the LCA Universal Interface (LUI), is based on four Xilinx XC3064-100 PGAs and extends

generic, programmable logic to the board level. This board, with suitable FPGA programming, is reused in multiple interface designs. The second board, the Device Universal Interface (DUI), is custom designed for each application. The DUI is usually quite simple since it only provides the electrical interface levels required by the target peripheral.

FPGA Strategy

Early in the design it was realized that goals for the project exceeded the capabilities of the LUI. In particular, the data transfer between the interface and the VTE DVSR occurs at a 40 MHz rate. This data stream needed to be multiplexed or demultiplexed between the 16 bit data path required by the DVSR and the 64 bit data path required by the Convex I/O system. Further, state machine operation at these speeds using the Xilinx devices installed on the LUI was considered too risky. This meant that some level of preprocessing would need to be performed on the DUI to reduce the demands on the LUI. Programmable logic was desired due to density and flexibility so a search was undertaken to find a family suitable to the task. The following requirements for the family were considered necessary:

Pin to pin delays on the order of 15-20 nS in order to generate the control signals necessary by the DVSR.

A gate count on the order of 2500 gates would allow implementation of everything considered necessary and still keep the design on a single board.

A reprogrammable technology that could be reconfigured in-circuit, to allow for ease of configuration between the two target peripherals.

Several FPGA families were considered for the design, however, at about this time Convex was selected by Lattice Semiconductor Corporation to be a beta site for their new pLSI family of nonvolatile FPGAs. The Special Systems group was already a user of the Lattice GAL devices due to their generic architecture and reprogrammability features, and had been privileged to input suggestions during the creation the the pLSI family. The emergence of the technology at the beginning of the design cycle was a stroke of good fortune; not only did the pLSI family satisfy all of the FPGA requirements, but it gave the Special Systems group an opportunity to evaluate a technology that offered many advantages.

DUI Architecture

The architecture required by the DUI to interface to either the VTE or DVS frame buffer is very straightforward (Figure 2). The original intent was to use the pLSI FPGAs to implement the high speed state machines required by the design. However, after some initial exposure to the pLSI, it quickly became apparent that the capabilities of the device far exceeded the original design requirements. As is often the case in situations such as this, the device capabilities suggested enhancements and the design began to evolve. The initial concept of the project was to develop two separate designs based on the same architecture. Separate configuration downloads utilizing

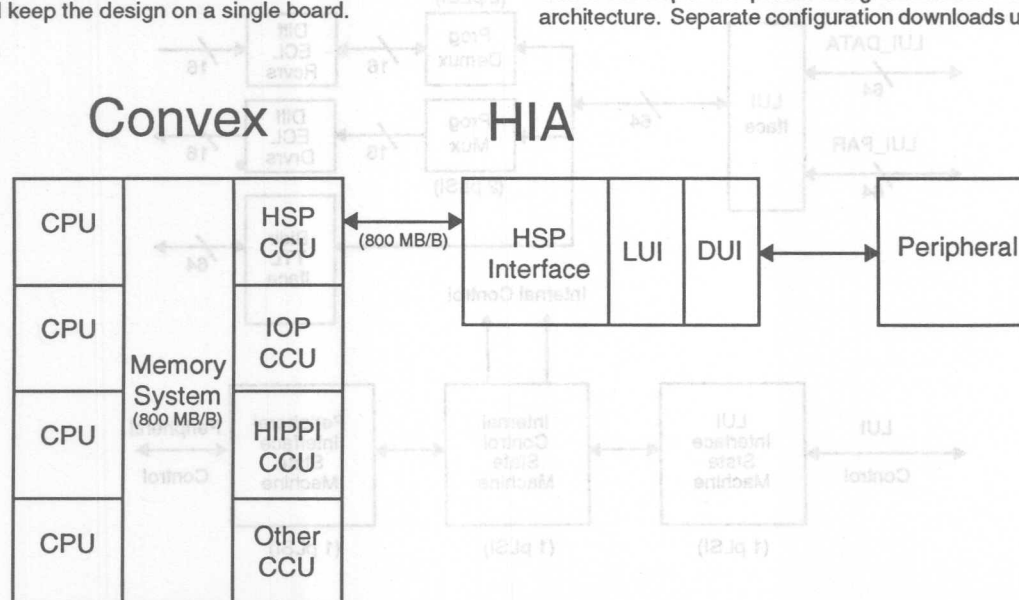


Figure 1. Convex System and I/O Structure

the in-circuit programmability feature would then configure the board to interface to whichever frame buffer was attached. However, it turned out that the pLSI1032, the first member of the pLSI family to be introduced, was sufficient to contain all state machines and control lines at the same time. A bit in a control register, set by the device driver, could be used to identify which frame buffer was attached, and thus determine which state machines and control lines would be active.

The realization that the design goals could be exceeded changed the focus of the design. Instead of limiting the interface to the original two frame buffers, the architecture began to shift to one which, with appropriate reprogramming of the FPGAs, might be used to accommodate other - as yet undetermined - frame buffers as well. This meant that the data path portion of the architecture needed to be as flexible as the control section. Instead of fixing the data path with discrete logic, it was decided to use FPGAs here also. This resulted in a data path that could handle alternate data byte ordering structures and parity schemes as well.

Design Implementation

Special Systems typically implements designs using the procedure shown in Figure 3. This procedure uses a

mixture of purchased CAD tools integrated with tools developed in-house. A model of the design is constructed using a hardware description language. This model is then subjected to rigorous testing which mimics the target environment. A mixture of schematic entry and logic synthesis translates the completed model into the final design. Gate level simulation of the result closes the design loop by verifying that the output acts the same as the original model. The result of this methodology is a design that requires a minimum of debugging in the lab.

Integration of the Lattice pLSI devices into this design cycle was a straightforward task. The pLSI development system required a slightly modified form of the Boolean equations already produced by the logic synthesis package. The only missing piece for this design was the ability to perform a gate level simulation on the final output. A tool to accomplish this in future designs is already in the planning stage.

Conclusions

The Lattice pLSI FPGA was a very favorable choice for the Frame Buffer Interface design. In particular, the speed and density of the device produced a design which would have been difficult to implement using slower FPGAs or discrete logic. Future Special Systems de-

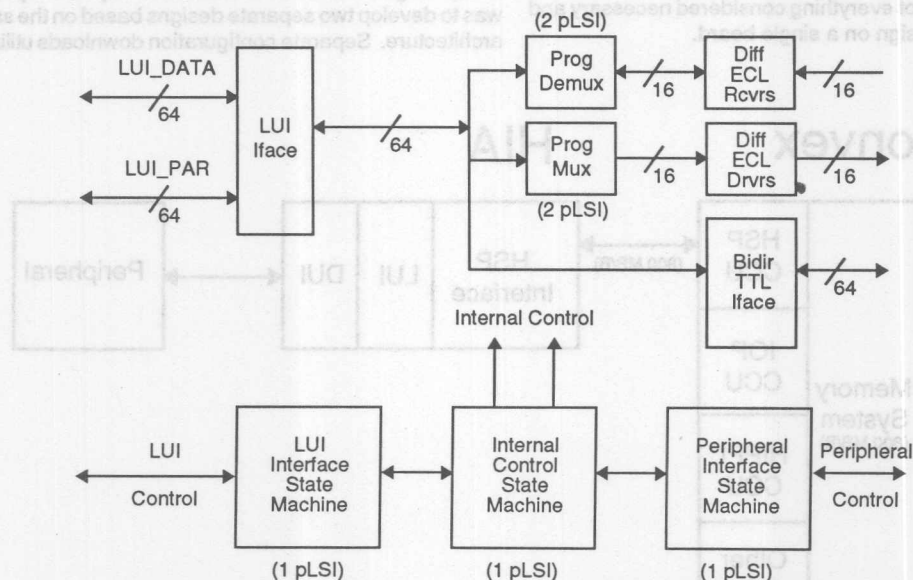


Figure 2. DUI Architecture

signs will be able to take advantage of the following features:

High speed operation which creates opportunities for designs previously beyond the reach of the LUI.

Enhanced routability and predictable routing delays that remove concerns of a small change adversely affecting the performance of a design.

In-circuit reprogrammability which allows design corrections or enhancements in the field with minimal disruption to the customers activities.

Features such as these will allow Convex to continue to produce designs which satisfy the ever increasing customer requirements.

References

[1] Lindsey, Lonnie, 1989, "High Definition Television: A Primer", Signal, August 1989.

[2] Convex Computer Corporation, 1990, "Supercomputers and High Definition TV".

[3] Lattice Semiconductor Corporation, 1991, "pLSI & ispLSI Design Guide".

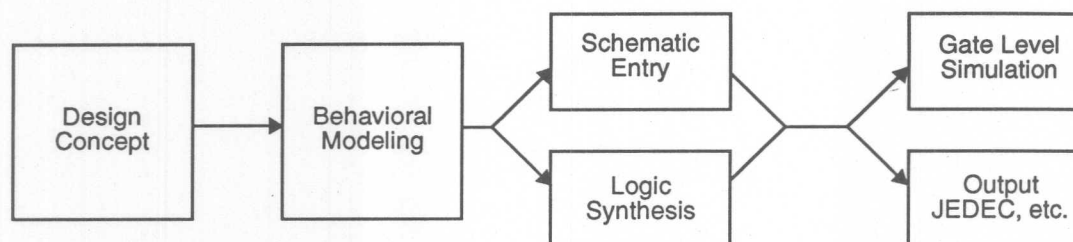


Figure 3. Design Methodology

Notes

- References
- [1] Lindsay, Connie, 1988, "High Definition Television: A Primer", Signal, August 1988.
 - [2] Convex Computer Corporation, 1990, "Supercomputer and High Definition TV".
 - [3] Lattice Semiconductor Corporation, 1991, "pLSI & pLSI Design Guide".

signs will be able to take advantage of the following features:

High speed operation which creates opportunities for designs previously beyond the reach of the LUT.

Enhanced routability and predictable routing delays that remove concerns of a small change adversely affecting the performance of a design.

In-circuit reprogrammability which allows design corrections or enhancements in the field with minimal disruption to the customers activities.

Features such as these will allow Convex to continue to produce designs which satisfy the ever increasing customer requirements.

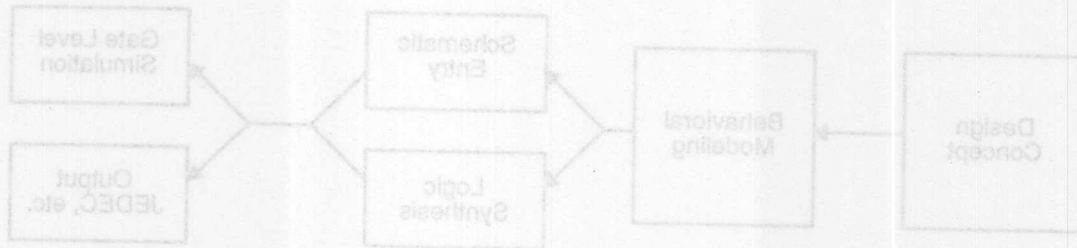


Figure 3. Design Methodology

Section 1: Introduction

Section 2: ispLSI and pLSI Architecture Overview

Section 3: ispLSI and pLSI Development Tools

Section 4: ispLSI and pLSI Application Notes

Section 5: GAL Architecture Overview

Section 6: GAL Development Tools

Section 7: GAL Application Notes

Section 8: In-System Programmable Generic Digital Switch (ispGDS)

Section 9: Design Techniques

Section 10: Article Reprints

Section 11: Technology, Quality, and Reliability Overview

Quality Assurance Program	11-1
Qualification Program	11-3
E ² CMOS Testability Improves Quality	11-5
Technology and Reliability	11-7
ISO 9000 Program	11-11

Section 12: General Section

Section 1: Introduction	
Section 2: i945L and i945L Architecture Overview	
Section 3: i945L and i945L Development Tools	
Section 4: i945L and i945L Application Notes	
Section 5: i945L Architecture Overview	
Section 6: i945L Development Tools	
Section 7: i945L Application Notes	
Section 8: In-System Programmable Generic Digital Switch (i945L)	
Section 9: Design Techniques	
Section 10: Article Page	
Section 11: Technology, Quality, and Reliability Overview	
Quality Assurance Program	11-1
Qualification Program	11-2
FEMOS Testability Improves Quality	11-3
Technology and Reliability	11-4
ISL 8000 Program	11-11
Section 12: General Section	

Quality Assurance Program

Introduction

Lattice views quality assurance as a corporate responsibility and an integral part of all operational activities. Lattice's Quality Assurance organization is independent from Manufacturing and has direct access to top management, assuring sufficient authority is afforded to quality issues.

Lattice's quality program is in full compliance to the quality assurance requirements of MIL-M-38510 Appendix A and all inspection system requirements of MIL-I-45208. Lattice is also fully certified to the ISO 9001 standard.

Reliability

All new products, processes and vendors must pass pre-defined evaluations before receiving initial qualification release. Major changes to products, processes or vendors require additional qualification before implementation. To assure continuing conformance to reliability goals, an ongoing monitor program is maintained on all products.

In-Process Control

Qualified product must be manufactured under strict quality controls that start with regulated procurement and documented inspection plans for all incoming materials. Sample testing and in-line monitoring as well as statistical process control charts provide constant feedback at each critical step of the manufacturing process.

Calibration

All equipment involved in determining product conformance to specifications through inspection, measurement or testing must be of the required accuracy. Equipment is calibrated and maintained on a defined interval against a nationally recognized standard. In addition, equipment must exhibit a suitable indicator showing calibration status as well as safeguards to disallow unauthorized adjustments.

Training

Key manufacturing personnel must complete a formal training program and obtain certification for each operation before they are allowed to perform activities affecting quality. Methods and records identifying the type and extent of training are maintained and recertification required on a yearly basis.

Subcontractor Control

All subcontracted manufacturing operations must be performed by sources exhibiting a quality program commensurate to that of Lattice. These suppliers are audited at least once a year to monitor their compliance to Lattice's quality initiatives and goals. Incoming inspection is performed to provide feedback and continuous improvement of subcontractor performance with the main objective being to control quality at the source. Communications and in-line data are continuously exchanged to allow real-time monitoring of subcontractor manufacturing operations.

Document Control

Every product and process must have adequate written documentation released and available at the point of use before production begins. All information related to the definition, manufacturing, testing and support of Lattice products or services shall be maintained and controlled. Initial release as well as subsequent changes must be properly reviewed and approved before implemented.

Nonconforming Material

Material found to be nonconforming to specified requirements is identified, segregated, analyzed and dispositioned per documented procedures. Records are maintained denoting the nature of the discrepancy as well as the final disposition. All dispositions involve the applicable engineering section and Quality Assurance. Where applicable, the root cause of the discrepancy will be identified and a corrective action implemented using the CAR (Corrective Action Request) form.

Failure Analysis

Failure modes discovered during qualification testing, inspections, customer returns or in-process screening are processed through Lattice's Failure Analysis group to determine the cause or relevancy of the failure. Verified failure modes are documented and corrective action initiated as required to eliminate the root cause.

Corrective Action

All operational functions utilize a documented corrective action system coordinated, recorded and monitored by Quality Assurance. The system is designed to provide for proactive problem identification and resolution in a timely manner. Inputs include vendor, internal and customer related problems. Emphasis is placed on effective elimination of the root cause to prevent recurrence of the problem.

Quality Assurance Program

Management is responsible for ensuring that employees have sufficiently well defined responsibilities, authority and organizational freedom to identify potential quality related problems as well as initiate and implement solutions.

Self Audit

Internal self audits of the entire quality and delivery system are performed per written procedures and to a predefined schedule. The functional audits evaluate actual method to written procedure. The results of these audits are documented on a checklist and any discrepancies are brought to the attention of personnel responsible for the audited area. Deficiencies require corrective actions must be initiated and subsequently verified as to deployment and effectiveness. A periodic review of these functional audit results and corrective actions is performed by Quality Assurance.

Procurement

All direct materials and services affecting quality or reliability of end product must be purchased from qualified sources. Selection of these critical suppliers is based upon one of more of the following: quality system audits, product qualification testing, correlation studies, incoming inspection and demonstrated ability. A qualified supplier list is maintained by Quality Assurance and used by Purchasing to control procurement. Each purchase order must specify the applicable controlling requirements for all such direct materials or services.

When applicable, the root cause of the discrepancy will be identified and a corrective action implemented using the CAR (Corrective Action Request) form.

Failure modes discovered during qualification testing, inspection, customer returns or in-process screening are processed through the Failure Analysis group to determine the cause or relevance of the failure. Verified failure modes are documented and corrective action initiated as required to eliminate the root cause.

All operational functions utilize a documented corrective action system coordinated, recorded and monitored by Quality Assurance. The system is designed to provide for proactive problem identification and resolution in a timely manner. Inputs include vendor, internal and customer related problems. Emphasis is placed on effective elimination of the root cause to prevent recurrence of the problem.

Introduction

Latice views quality assurance as a corporate responsibility and an integral part of all operational activities. Latice's Quality Assurance organization is independent from Manufacturing and has direct access to top management, assuring sufficient authority is afforded to quality issues.

Latice's quality program is in full compliance to the quality assurance requirements of MIL-M-38510 Appendix A and all inspection system requirements of MIL-14820B. Latice is also fully certified to the ISO 9001 standard.

Reliability

All new products, processes and vendors must pass pre-defined evaluations before receiving initial qualification release. Major changes to products, processes or vendors require additional qualification before implementation. To assure continuing conformance to reliability goals, an ongoing monitor program is maintained on all products.

In-Process Control

Qualified product must be manufactured under strict quality controls that start with regulated procurement and documented inspection plans for all incoming materials. Sample testing and in-line monitoring as well as statistical process control charts provide constant feedback at each critical step of the manufacturing process.

Calibration

All equipment involved in determining product conformance to specifications through inspection, measurement or testing must be of the required accuracy. Equipment is calibrated and maintained on a defined interval against a nationally recognized standard. In addition, equipment must exhibit a suitable indicator showing calibration status as well as safeguards to disallow unauthorized adjustments.

Training

Key manufacturing personnel must complete a formal training program and obtain certification for each operation before they are allowed to perform activities affecting quality. Method and records identifying the type and extent of training are maintained and recertification re-evaluated on a yearly basis.

Qualification Program

Introduction

Lattice has an intensive qualification program for examining and testing new products, processes, and vendors in order to insure the highest levels of quality. Lattice's Quality and Reliability Group is responsible for defining and implementing this qualification program.

The following table outlines the steps which must be performed before a new product, package or process is released. The requirements listed below are general guidelines. Detailed information on Lattice's qualification process is available to customers upon request.

Qualification Requirements

Test	# of Samples	Duration		
		New Product	New Wafer Process	New Package
125° C Operating Lifetest (5.25V)	300	1,000 Hours	2,000 Hours	2,000 Hours ¹
150° C Biased Retention Bake (5.25V)	450	1,000 Hours	2,000 Hours	2,000 Hours ¹
Endurance Cycling	75	10,000 Cycles	10,000 Cycles	N/A
ESD (CDM, HBM, MM)	216	End of Test	End of Test	N/A
Latch-Up Immunity	27	End of Test	End of Test	N/A
Temperature Cycling (-65 to 150° C)	150	1,000 Cycles	1,000 Cycles	1,000 Cycles
Biased 85/85 (5V)	225	N/A	1,000 Hours	1,000 Hours
Autoclave (121° C, 15psig)	150	N/A	336 Hours	336 Hours
Lead Integrity (DIP only)	9	N/A	N/A	End of Test
Solderability	9	N/A	N/A	End of Test
Physical Dimensions	6	N/A	N/A	End of Test

1. Required for new assembly technologies only.

Qualification Program

Reliability Monitor Program

The Reliability Monitor Program provides for a periodic reliability monitor of Lattice products. The program assures that all Lattice products comply on a continuing basis with established reliability and quality levels.

The Reliability Monitor Program is designed to monitor all fab and assembly facilities as well as each process technology in production. A summary of the program test and sampling plan is shown below.

Short Term Process Monitor (Bi-Weekly)

Test	# of Samples	Duration
125° C Operating Lifetest (6.50V)	70	160 Hours
150° C Biased Retention Bake (5.25V)	70	160 Hours
Autoclave (121° C, 15psig)	35	160 Hours

Long Term Process Monitor (Monthly)

Test	# of Samples	Duration
125° C Operating Lifetest (6.00V)	100	2000 Hours
150° C Biased Retention Bake (5.25V)	150	2000 Hours

Ongoing Package Monitor (Monthly)

Test	# of Samples	Duration
Temperature Cycling (-65 to 150° C)	50	1000 Cycles
85° C / 85% RH	75	2000 Hours

E²CMOS Testability Improves Quality

Introduction

The inherent testability of Lattice's E²CMOS PLDs significantly improves their quality and reliability. By using electrically erasable EEPROM technology to produce GAL, pLSI and ispLSI devices, Lattice is able to perform 100% AC/DC, functional, and parametric testing of every single device. In order to achieve the highest quality levels, Lattice programs and tests each device repeatedly throughout the manufacturing process.

Actual Test vs. Simulated Test

Why is "actual test" so significant? PLDs, unlike most other semiconductor devices, have a programmable element that determines the final device functionality and AC/DC performance. These programmable elements can be fabricated from metal link fuses, programmable diodes or transistors, volatile static RAM cells, UV EPROM cells or electrically erasable EEPROM cells. Each of these technologies carries a different variability of programming success and a variance in the impact of the programming success on the performance and reliability of the device.

The most common programmable elements are the metal fuse, EPROM cell and EEPROM cell. Of these element types, only the EEPROM cell can be thoroughly tested by the manufacturer prior to shipment to an end user OEM.

EEPROM Allows Actual Test

Each of the technologies identified above can be programmed. In this manner they are all the same. The differences become apparent when the erase times are analyzed. Metal link and One-Time Programmable (OTP) devices cannot be erased. UV EPROM devices can be erased, however the time required is 20-30 minutes (in an expensive windowed package). EEPROM devices, on the other hand, offer instant erasability on the order of 50 ms (thousandth's of a second). The advantage of this instant erase for manufacturing test is significant. Instant erase allows instant re-patterning for additional testing.

EEPROM technology has been used for PLD manufacturing by Lattice for more than a decade. Lattice refers to their high performance EEPROM technology as E²CMOS technology. Extensive reliability studies of the technology have been performed with industry-wide acceptance, including the military.

Other Methods Are Imprecise

All PLD devices must be tested to some degree to validate functionality and performance. Technologies that are not erasable or require lengthy erase times severely constrain the test flexibility. Since the normal "user" programmable elements cannot be programmed during manufacture (all elements must be available for end-user programming) the manufacturers of one-time programmable PLDs resort to using simulated and correlated performance of test rows, test columns and phantom or dummy-test arrays. At best, this is a statistical measure of the actual device performance. One need only look at the "normal" programming yield fallout of 0.5 to 3% or the "acceptable" post-programming test vector and board yield fallout of 0.5 to 2% to know that this correlation is weak. The quality systems of today are measuring defects in parts per million (PPM). A six sigma program requires less than 3.4 PPM, four orders of magnitude less than that achievable with non-testable PLDs.

Actual Matrix Patterning

The unique capability of E²CMOS devices to be instantly electrically erased allows these devices to be patterned multiple times during Lattice's manufacturing test. Normal array cells in the programmable matrix are patterned, erased and tested again and again. The test rows or columns, phantom arrays, etc., that are used with other technologies are not necessary with E²CMOS devices. Programmability of every cell is checked dozens of times.

Historically, the checking of a successful programming operation consisted of no more than a pass/fail verification step. This digital, go/no go check is not adequate to assure that the cell is programmed properly with sufficient margin to guarantee long-term reliable performance of the device. Lattice E²CMOS devices are processed through a proprietary cell verification step that consists of an analog measure (to millivolt accuracy) of the actual cell threshold. This capability is used for extensive reliability and quality measurements and testing.

Worst Case AC/DC Testing

A PLD does not have a defined function until the engineer patterns the device with his custom pattern. The manufacturer, when considering the testing of a PLD, must consider the hundreds of different architecture and functional variations that can be created by the end user. Each configuration of architecture brings on a different set of worst case pattern and stimulus conditions. Quick application of a series of worst case patterns that cover

E²CMOS Testability Improves Quality

all of the permutations of input combinations, array load and switching, and output configuration is required.

E²CMOS devices offer instant erasability to address this reconfiguration and test problem. Testing each additional worst case configuration takes fractions of a second, allowing multiple patterns to be checked to assure performance to rated speeds. The final result is a device with defects reduced from PPH (parts per hundred) to PPM (parts per million).

During manufacture (all elements are programmed), the manufacturer of one-time programmable PLDs resort to using simulated and correlated performance of test rows, test columns and graininess or dummy-test arrays. At best, this is a statistical measure of the actual device performance. One need only look at the "normal" programming yield (about 0.5 to 0.8% of the "acceptable" post-programming test vector and board yield) of 0.5 to 0.8% to know that the testing is weak. The quality systems of today are measuring defects in parts per million (PPM). A six sigma program requires less than 3.4 PPM (four orders of magnitude less than that achievable with non-testable PLDs).

Actual Matrix Patterning

The unique capability of E²CMOS devices to be internally electrically erased allows these devices to be patterned multiple times during Lattice's manufacturing test. Non-metal array cells in the programmable matrix are patterned, erased and tested again and again. The test rows or columns, phantom arrays, etc., that are used with other technologies are not necessary with E²CMOS devices. Programmability of every cell is checked dozens of times.

Historically, the checking of a successful programming operation consisted of no more than a pass/fail verification step. This digital, go/no go check is not adequate to assure that the cell is programmed properly with sufficient margin to guarantee long-term reliable performance of the device. Lattice E²CMOS devices are processed through a proprietary cell verification step that consists of an analog measure (to millivolt accuracy) of the actual cell threshold. This capability is used for extensive reliability and quality measurements and testing.

Worst Case ACDC Testing

A PLD does not have a defined function until the engineer patterns the device with his custom pattern. The manufacturer, when considering the testing of a PLD, must consider the hundreds of different architectures and functional variations that can be created by the end user. Each configuration of architecture brings on a different set of worst case pattern and stimulus conditions. Quick application of a series of worst case patterns that cover

The inherent testability of Lattice's E²CMOS PLDs significantly improves their quality and reliability. By using electrically erasable EEPROM technology to produce GAL, PLD and ISP devices, Lattice is able to perform 100% ACDC, functional, and parametric testing of every single device. In order to achieve the highest quality levels, Lattice programs and tests each device repeatedly throughout the manufacturing process.

Actual Test vs. Simulated Test

Why is "actual test" so significant? PLDs, unlike most other semiconductor devices, have a programmable element that determines the final device functionality and ACDC performance. These programmable elements can be fabricated from metal link fuses, programmable diodes or transistors, volatile static RAM cells, UV EPROM cells or electrically erasable EEPROM cells. Each of these technologies carries a different variability of programming success and a variance in the impact of the programming success on the performance and reliability of the device.

The most common programmable elements are the metal link, EPROM cell and EEPROM cell. Of these element types, only the EEPROM cell can be thoroughly tested by the manufacturer prior to shipment to an end user OEM.

EEPROM Allows Actual Test

Each of the technologies identified above can be programmed. In this manner they are all the same. The differences become apparent when the erase times are analyzed. Metal link and One-Time Programmable (OTP) devices cannot be erased. UV EPROM devices can be erased, however the time required is 20-30 minutes (in an expensive windowed package). EEPROM devices, on the other hand, offer instant erasability on the order of 30 ms (thousandths of a second). The advantage of this instant erase for manufacturing test is significant. Instant erase allows instant re-patterning for additional testing.

EEPROM technology has been used for PLD manufacturing by Lattice for more than a decade. Lattice refers to their high performance EEPROM technology as E²CMOS technology. Extensive reliability studies of the technology have been performed with industry-wide acceptance, including the military.

Technology and Reliability

Introduction

Lattice maintains a comprehensive reliability qualification program to assure that each product achieves its reliability goals. After initial qualification, data is continuously accumulated through monitor programs so as to further drive failure rates down. Each product's qualification plan is generated in conformance to Lattice's Qualification Policy with failure analysis in conformance to Lattice's Failure Analysis Procedures. Both documents are contained in Lattice's Quality Assurance Manual, which can be obtained upon request. Failure rates in this reliability summary are expressed in FITS. Due to the very low failure rate of integrated circuits, it is convenient to refer to failures in a population during a period of 10^9 device hours; one failure in 10^9 device hours is defined as one FIT.

Process Overview

Lattice Semiconductor is using the fourth and fifth generation of its advanced UltraMOS® process for its current manufacturing (Figure 1).

Basic Theory of Operation

An E²CMOS cell is built around a MOS transistor with a floating gate which is externally charged or discharged by a small programming current. If the floating gate is charged up to a positive potential by removing electrons from the floating gate, the cell transistor is turned on,

storing a binary zero in the cell. If the floating gate is charged to a negative potential by placing electrons on the floating gate, the transistor is kept in the non-conducting or off state, which writes a binary one into the cell. In addition to the floating gate or sense device, an additional select transistor, or pass gate, is added in series with the cell to isolate it from the array during read and write operations. A schematic representation of this cell is shown in Figure 2. In addition to the conventional bit line and word line, the E²CMOS cell also has an additional line for the matrix control gate (MCG) which controls the potential of the floating gate.

The cell is programmed by applying a programming pulse to either the matrix control gate or the bit line of a cell which has been selected by an applied high voltage on the word line. Programming takes place when electrons tunnel through the thin tunneling dielectric shown in the schematic by the small notch in the floating gate over the drain of the sense device. Before describing the detailed operation of the E²CMOS cell, the requirements and tradeoffs of the process technology will be reviewed.

E²CMOS Process Technology

Lattice's E²CMOS technology is based upon a highly successful combination of CMOS and NMOS technologies. The requirements for both on-chip high voltage and high speed devices put severe restrictions on the process technology. By incorporating both pumped substrate

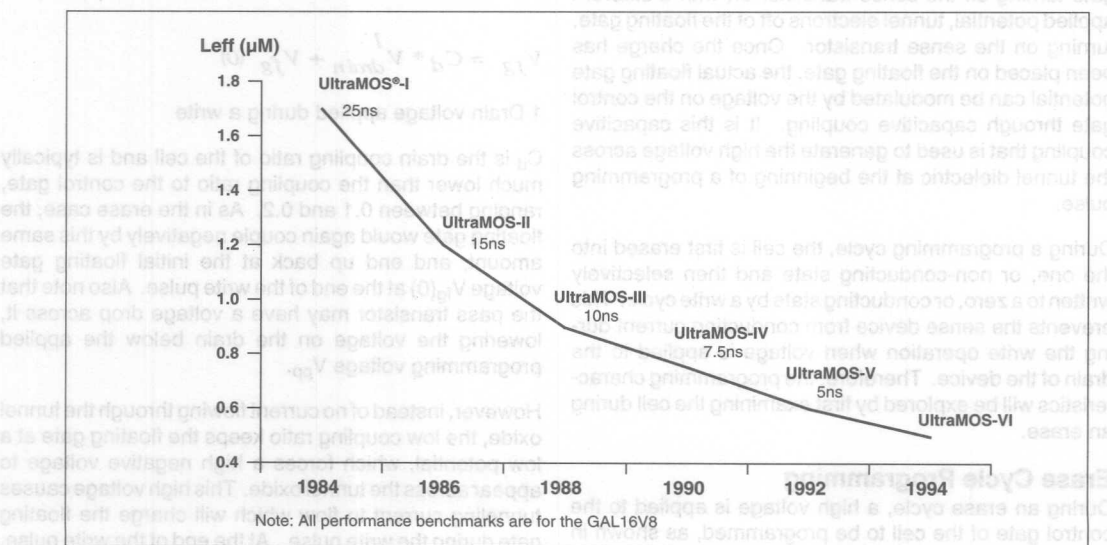


Figure 1

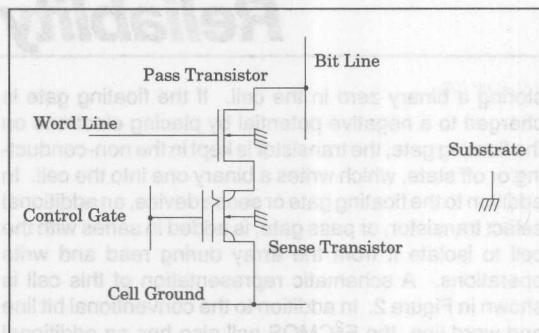


Figure 2

techniques and depletion devices from NMOS technology with low power CMOS devices, Lattice's E²CMOS technology maintains high performance while meeting the high voltage requirements of programming.

In addition to combining the techniques of both NMOS and CMOS, Lattice's E²CMOS technology incorporates an ultra-clean, ultra-thin tunneling oxide approximately 100 Angstroms thick. The requirements placed on these oxides will be much more apparent after the programming characteristics of the cell are examined.

Single Cell Programming

The E²CMOS cell is programmed by placing a high voltage across the thin tunnel dielectric. The resulting tunneling current will tunnel electrons onto the floating gate turning off the sense transistor or, with a different applied potential, tunnel electrons off of the floating gate, turning on the sense transistor. Once the charge has been placed on the floating gate, the actual floating gate potential can be modulated by the voltage on the control gate through capacitive coupling. It is this capacitive coupling that is used to generate the high voltage across the tunnel dielectric at the beginning of a programming pulse.

During a programming cycle, the cell is first erased into the one, or non-conducting state and then selectively written to a zero, or conducting state by a write cycle. This prevents the sense device from conducting current during the write operation when voltage is applied to the drain of the device. Therefore, the programming characteristics will be explored by first examining the cell during an erase.

Erase Cycle Programming

During an erase cycle, a high voltage is applied to the control gate of the cell to be programmed, as shown in Figure 2. If all current through the tunnel oxide is

neglected, the floating gate will simply track the applied voltage following the relationship of a capacitive divider, where C_{up} is the coupling ratio of the cell, and is typically between 0.7 and 0.8. At the end of the erase pulse, the floating gate would again couple negatively by the same amount, and end up back at the initial floating gate voltage $V_{fg}(0)$.

$$V_{fg}^1 = C_{up} * V_{cg}^2 + V_{fg}^3 (0)$$

1 floating gate voltage; 2 Control gate voltage; 3 Initial floating gate voltage

However, the high voltage applied across the tunnel dielectric causes tunneling current to flow, which will discharge the floating gate during the erase pulse. At the end of the erase pulse, the floating gate will end up at a potential that is lower than the initial floating gate voltage by the amount that the floating gate has decayed during the pulse. This negative voltage is sufficient to turn off the sense transistor during a read operation. The magnitude of the control gate voltage which is required to couple this negative floating gate voltage up to the threshold of the sense device and actually turn it on after the erase pulse is defined as the programmed high threshold V_{thHigh} .

Write Cycle Programming

During the write cycle, a high voltage is applied to the bit line of the cell to be programmed. If all current through the tunnel oxide is again neglected, the floating gate will track the applied drain voltage following the relationship of a capacitive divider:

$$V_{fg} = C_d * V_{drain}^1 + V_{fg} (0)$$

1 Drain voltage applied during a write

C_d is the drain coupling ratio of the cell and is typically much lower than the coupling ratio to the control gate, ranging between 0.1 and 0.2. As in the erase case, the floating gate would again couple negatively by this same amount, and end up back at the initial floating gate voltage $V_{fg}(0)$ at the end of the write pulse. Also note that the pass transistor may have a voltage drop across it, lowering the voltage on the drain below the applied programming voltage V_{pp} .

However, instead of no current flowing through the tunnel oxide, the low coupling ratio keeps the floating gate at a low potential, which forces a high negative voltage to appear across the tunnel oxide. This high voltage causes tunneling current to flow which will charge the floating gate during the write pulse. At the end of the write pulse,

the floating gate will end up at a potential that is higher than the initial floating gate voltage by the amount that the floating gate has charged during the pulse. This positive voltage is sufficient to turn on the sense transistor during a read operation. The magnitude of the control gate voltage which is required to couple this positive floating gate voltage down to the threshold of the sense device and actually turn it off is defined as the programmed low threshold V_{tLow} .

Reading the Cell

After an erase cycle the charge on the floating gate has left the sense transistor in the off, or non-conducting "1" state. If the erase cycle is followed by a write cycle, then the floating gate charge leaves the sense device in the on or conducting "0" state. Therefore, the data in the cell can be read by simply sensing the cell current when biased with the control gate centered between the on and off states. This is shown schematically in Figure 3. The bit line and control gate voltages are selected to minimize the potential across the tunnel dielectric during a read in order to maximize the retention of the floating gate charge. The actual magnitude of the programmed thresholds, and thus the margins of the cell, are controlled by the programming voltage, the physical cell layout, and the tunnel oxide electrical characteristics. The key electrical properties of the tunnel oxide will be examined since they are critical in determining both the programming properties and the reliability of the E²CMOS cell.

Tunnel Oxide Electrical Characteristics

The E²CMOS cell is programmed by placing a high voltage across the thin tunnel dielectric. The tunnel oxide is sufficiently thin, typically with a thickness between 80 and 120 Angstroms, that electrons will tunnel through the dielectric and program the cell. The exact nature of the tunneling mechanism is important because in addition to determining the amount of voltage required to get sufficient current through the oxide to program the cell, the tunnel characteristic also must be a very strong function of voltage to prevent the charge from leaking off of the floating gate during the low voltage, normal read, operation.

In addition to the electrical current voltage characteristics, thin tunneling dielectrics must also be characterized by the amount of charge that can pass through the oxide without altering its electrical properties. Electron traps located in the oxide will capture some of the electrons passing through the dielectric. As this trapped charge builds up in the oxide, the electrical properties change, and eventually the oxide wears out and ruptures. Thus, in addition to controlling the erase and write characteristics of an E²CMOS cell, the tunnel oxide, and oxide

quality, play a major role in the reliability of the technology.

I-V Characteristics of Thin Tunnel Dielectrics

A typical I-V characteristic of a thin tunnel oxide is shown in Figure 4. Since the current must flow through this oxide in both directions, the characteristic of the oxide is shown for both positive and negative polarities. Note that a higher negative voltage is required to get the same current as in the positive voltage case because of an additional voltage drop that occurs across a depletion region formed in the silicon for negative applied voltage. For a tunnel oxide of approximately 100 Angstroms in thickness, the maximum voltage developed during programming is roughly 10 volts, or a field strength of 10 MV/Cm. This very high applied field stress, needed for the tunneling process requires very high quality oxides and very clean processing conditions. Optimizing the thickness of the tunneling dielectric and trading off between the programming characteristics and the oxide reliability is a requirement of the E²CMOS technology.

The I-V characteristic is a very strong function of oxide thickness and follows the relationship of the Fowler-Nordheim tunneling equation, where A and B are the Fowler-Nordheim coefficients.

$$I_{tox}(V) = A * Area * \left(\frac{V^2}{Tox^2} \right) * Exp \left(\frac{-B * Tox}{V} \right)$$

This equation can be rewritten in terms of the field across the oxide as below, which is independent of oxide thickness.

$$I_{tox}(E) = A * Area * E^2 * Exp \left(\frac{-B}{E} \right)$$

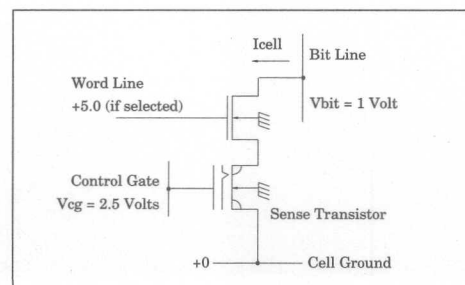


Figure 3

Technology and Reliability

Charge-to-Breakdown of Thin Tunnel Oxides

The I-V characteristic shown in the previous section was measured at sufficiently low current densities such that no charge trapping occurred during the measurements. If, however, a large amount of charge is passed through the oxide, the trapped charge in the oxide will alter the electrical I-V characteristic. The increase in voltage required to get the same tunneling current after a large amount of charge passes through the oxide will reduce the amount of charge transferred into the cell during a programming cycle and therefore reduce the cell programming margins with continued cycling. The magnitude of the charge required to shift the I-V characteristic depends on the quality and the number of traps in the oxide.

In addition to this shift in the electrical properties of the tunnel dielectric, defects in the oxide, whose properties change as the charge passes through the oxide, will actually cause the oxide to rupture after a finite amount of charge has passed through the dielectric. The maximum charge that can be passed through the oxide prior to oxide breakdown, or the oxide fluence expressed in Coulombs/Cm², can be determined by passing current through a tunnel dielectric until it ruptures. This physical limitation on the current that can be passed through the tunnel oxide places a limit on the number of programming cycles that can be performed on any E² device. This cycling limit, or endurance, is dependent on the quality of the tunnel dielectric and its associated defect density as well as the exact programming stress on the oxide. Lattice's technology maximizes the endurance of the devices through careful control of the requirements on the oxide as well as by optimizing the quality of the dielectric.

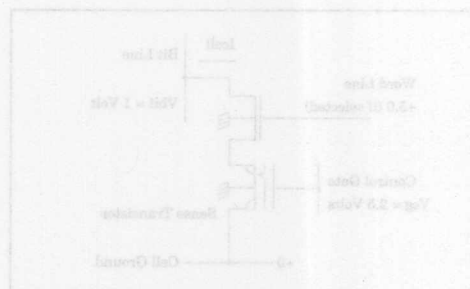


Figure 3

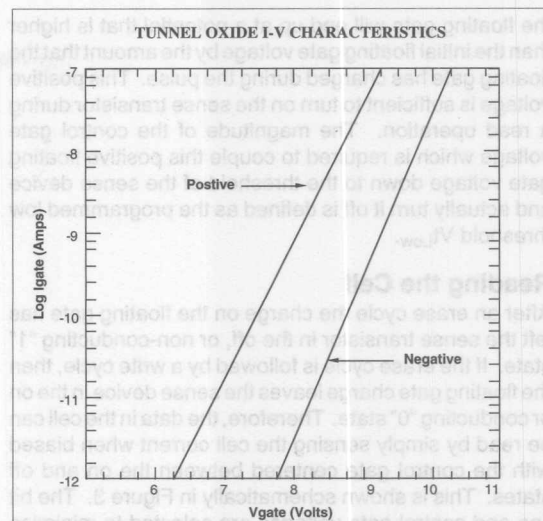


Figure 4

Tunnel Oxide Electrical Characteristics
The E²CMOS cell is programmed by placing a high voltage across the thin tunnel dielectric. The tunnel oxide is sufficiently thin (typically with a thickness between 50 and 120 Angstroms) that electrons will tunnel through the dielectric and program the cell. The exact nature of the tunneling mechanism is important because in addition to determining the amount of voltage required to get sufficient current through the oxide to program the cell, the tunnel characteristics also must be a very strong function of voltage to prevent the charge from leaking off of the floating gate during the low voltage, normal read operation.

In addition to the electrical current voltage characteristics, thin tunneling dielectrics must also be characterized by the amount of charge that can pass through the oxide without altering its electrical properties. Electron traps located in the oxide will capture some of the electrons passing through the dielectric. As this trapped charge builds up in the oxide, the electrical properties change, and eventually the oxide wears out and ruptures. Thus, in addition to controlling the stress and with characteristics of an E²CMOS cell, the tunnel oxide, and oxide

ISO 9000 Program

Introduction

Lattice is proud to be the first major PLD manufacturer to achieve ISO 9000 certification. Lattice Quality Systems have been certified, and the company is registered to the ISO 9000 standard. Lattice certification is for ISO 9001, the most comprehensive of the various ISO 9000 levels, covering the design, manufacturing, sales, and service functions.

ISO 9000 Certification

Certification to the ISO 9000 standard provides a recognized and standardized basis for the continued development of the quality and reliability of Lattice products. This certification assures Lattice's customers that its Quality Systems are well organized and embody a "Quality First" philosophy. It also reaffirms Lattice's promise to provide its customers with the highest quality and most reliable products in the industry.

What is ISO 9000?

The ISO 9000 series is an international version of British Standard BS 5750, intended to define the quality management systems for a wide range of an organization's

activities. The standard was initiated by the British Standards Institution, which over the last 80 years has certified over 9,000 Quality Systems. Today, both the CEN (European Committee for Standardization), which is commissioned to coordinate quality standards in Europe and remove potential trade restrictions within and outside the European Community, and the USA Standard ANSI/ASQC have adopted the ISO 9000 series.

Four quality standards make up the ISO 9000 series: ISO 9004, ISO 9003, ISO 9002, and ISO 9001. ISO 9004 is an informational document containing guidelines for Quality Management and Quality Systems. ISO 9003 guarantees quality in a product's final testing and inspection. ISO 9002 confirms quality in the production and installation of a product. ISO 9001 assures quality in a product's design, development, production, and installation. ISO 9001 is composed of 20 system sections, including the ISO 9002 and ISO 9003 subsets. Lattice is certified to the most comprehensive quality standard of the series, ISO 9001, and registered with the American Society for Quality Control's Registration Accreditation Board.



Lattice Semiconductor: First PLD Supplier to Achieve ISO 9000 Certification

activities. The standard was initiated by the British Standard Institution, which over the last 80 years has certified over 3,000 Quality Systems. Today, both the CEN (European Committee for Standardization), which is commissioned to coordinate quality standards in Europe, and various potential trade restrictions within and outside the European Community, and the USA Standard ANSI ASQC have adopted the ISO 9000 series.

Four quality standards make up the ISO 9000 series: ISO 9001, ISO 9002, ISO 9003, and ISO 9004. ISO 9001 is an international document containing guidelines for Quality Management and Quality Systems. ISO 9002 guarantees quality in a product's final testing and inspection. ISO 9003 confirms quality in the production and installation of a product. ISO 9004 assures quality in a product's design, development, production, and installation. ISO 9004 is composed of 20 system sections, including the ISO 9001 and ISO 9003 subparts. Lattice is certified to the most comprehensive quality standard of the series, ISO 9001, and registered with the American Society for Quality Control's Registration Accreditation Board.

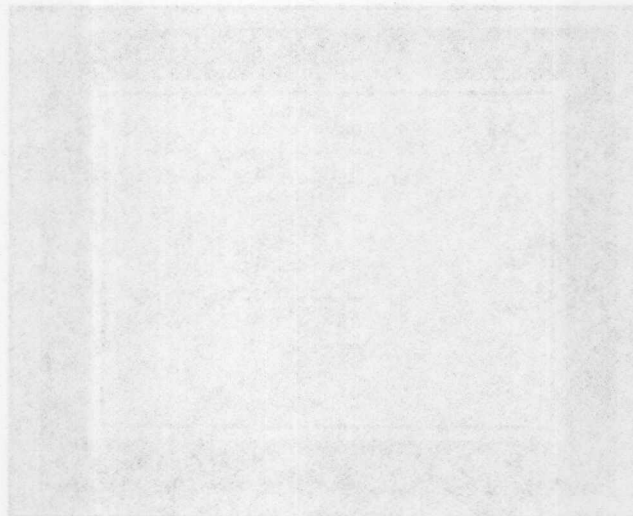
Lattice is proud to be the first major PLD manufacturer to achieve ISO 9000 certification. Lattice Quality Systems have been certified, and the company is registered to the ISO 9000 standard. Lattice certification is for ISO 9001, the most comprehensive of the various ISO 9000 levels, covering the design, manufacturing, sales, and service functions.

ISO 9000 Certification

Certification to the ISO 9000 standard provides a recognized and standardized basis for the continued development of the quality and reliability of Lattice products. This certification assures Lattice's customers that its Quality System is well organized and embodies a "Quality First" philosophy. It also reaffirms Lattice's promise to provide its customers with the highest quality and most reliable products in the industry.

What is ISO 9000?

The ISO 9000 series is an international version of British Standard BS 5750, intended to define the quality management systems for a wide range of an organization's



Lattice Semiconductor First PLD Supplier to Achieve ISO 9000 Certification

Section 1: Introduction

Section 2: ispLSI and pLSI Architecture Overview

Section 3: ispLSI and pLSI Development Tools

Section 4: ispLSI and pLSI Application Notes

Section 5: GAL Architecture Overview

Section 6: GAL Development Tools

Section 7: GAL Application Notes

Section 8: In-System Programmable Generic Digital Switch (ispGDS)

Section 9: Design Techniques

Section 10: Article Reprints

Section 11: Technology, Quality, and Reliability Overview

Section 12: General Section

Lattice Bulletin Board Systems	12-1
Cost of Ownership: An Overview	12-7
Hidden Costs in PLD Usage	12-9
ISP: Winning at the Bottom Line	12-15
Gate Array and High Density PLD Cost Analysis	12-17
Sales Offices	12-19

12-19	Sales Offices
12-17	Gate Array and High Density PLD Cost Analysis
12-16	ICP: Winning at the Bottom Line
12-15	Hidden Costs in PLD Usage
12-9	Cost of Ownership: An Overview
12-7	Office Bulletin Board Systems
12-1	Section 12: General Section

Section 11: Technology, Quality, and Reliability Overview

Section 10: Article Reprints

Section 9: Design Techniques

Section 8: In-System Programmable Generic Digital Switch (iagDS)

Section 7: GAL Application Notes

Section 6: GAL Development Tools

Section 5: GAL Architecture Overview

Section 4: iapLSI and pLSI Application Notes

Section 3: iapLSI and pLSI Development Tools

Section 2: iapLSI and pLSI Architecture Overview

Section 1: Introduction

Lattice Bulletin Board Systems

Introduction

Lattice maintains two Bulletin Board Systems (BBSs) to communicate with customers. One BBS is located in Milpitas, California at Lattice's Silicon Valley Design Center. This BBS provides primary ispLSI and pLSI support. The second BBS is located in Hillsboro, Oregon at Lattice's headquarters. This BBS provides primary GAL support and secondary ispLSI and pLSI support. The following two sections explain in detail how to connect to each of these BBSs and how to transfer information.

Using the Lattice Silicon Valley BBS

The Silicon Valley BBS is for ispLSI and pLSI customers, distributors and FAEs who are requesting technical support on our ispLSI and pLSI families of HDPLDs. You can use the Silicon Valley BBS to:

- Transfer designs to and from Lattice Application Engineers
- Join conferences to share and exchange information with Lattice Application Engineers and other users

Telephone number and Communication Software Setup

The telephone number for the Silicon Valley BBS is (408) 428 - 6417. The BBS supports modem speeds from 300-9600 Baud, and supports the typical default

communication parameters of eight data bits, one stop bit, and no parity (8-N-1).

New User

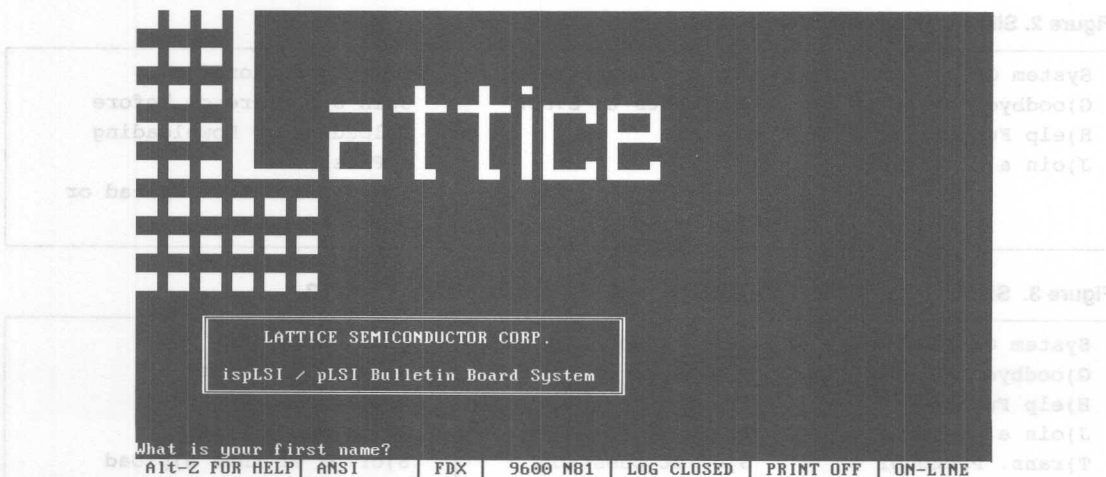
If you have not used the Silicon Valley BBS before, the system will first ask if you want the graphics mode. This mode will help a first time user by displaying different options by either blinking or displaying a different color text, if you have a color display.

You will then be asked for your first name, your last name and password (see figure 1). The user name must be your name; do not use your company name as a user name. The password can be up to 12 alphanumeric characters long. You will also be prompted to fill out a short script. You should be prepared with information about the Lattice software you are using and the 10 digit serial number from the Lattice security block(s).

After you complete the questionnaire, the system will display the main menu. As a first time user of the BBS, you have no rights to upload or download files. Your security level must be upgraded. This is done three times a day: Monday through Friday at 8:00 a.m., 12:00 noon and 5:00 p.m. except on holidays. All times are Pacific Standard Time (PST) or Pacific Daylight Savings Time (PDT).

For subsequent access, after you have logged on to the system, you will be asked if you want to scan for messages. Answer (Y/N) and press the enter key.

Figure 1. Silicon Valley BBS Initial Screen



Lattice Bulletin Board Systems

Main Menu

From this point, you'll be at the Main menu, with access to all other menus. Note that the Join option is not available to a new user until your security level has been upgraded.

Listed below is a brief description of the options available to a new user

- G - Hang up
- H - Help menu - Command options and description
- C - Leave a message to the System Administrator (SYSOP)

For users who have been upgraded and have previously joined a conference, the figure 3 menu will be seen. This menu has seven additional commands that are accessible. When you login to the BBS and have previously joined a conference, the menu in figure 3 will be the main menu. The conference you are in is the location you will be placed during your next successful login. Your options will be:

- D - Download a file from the BBS (Instruct the BBS computer to go into send mode)
- E - Leave a message for another user
- J - Join conference - Change to conference 1 or 2
- R - Read a message left by another user
- S - Required questionnaire about design, before Upload is started

- T - Transfer protocol - user specified type of file transfer method

- U - Upload a file - Instruct the BBS computer to go into receive mode

Upload files to the BBS

If you need to upload (send) a file to the Lattice BBS, the file should be zipped up, and a readme file should be added to the zip file. The zip utilities compress the file size and help to eliminate file transmission errors. The readme file should have a description of the questions, comments and/or problem you have.

To upload a file to the BBS, do the following:

1. Type J)oin 1 or 2 <enter> - This puts you into conference 1 or 2.
2. Type T)ransfer protocol <enter>.
3. Select which file transfer method you want - (X, Y, or Z protocol).
4. Type S)cript <enter> and fill out the questionnaire about your design and the design tools used.
5. Type U)pload *filename* <enter>. *Filename* is the name it will be called on the BBS.
6. Select the send file utility on your software package. If you are using a Procomm like software package, you press the "Page Up" key.

Download files from the BBS

To download a file to the BBS, do the following:

Figure 2. Silicon Valley BBS Main Menu

System Operations G)oodbye (Hang Up) H)elp Functions J)oin a Conference	Message Operations C)omments to SYSOP	File Operations Join a conference before Uploading or Downloading a File New Users cannot Upload or Download Files
--	--	---

Figure 3. Silicon Valley BBS Menu Selections available in conference 1 and 2

System Operations G)oodbye (Hang Up) H)elp Functions J)oin a Conference T)rans. Protocol	Message Operations C)omments to SYSOP R)ead Message S)cript Question	File Operations D)ownload a File E)nter a Message To Upload a File: S)cript #1 then U)pload
--	---	---

Lattice Bulletin Board Systems

1. Type J)oin 1 or 2 <enter> - This puts you into conference 1 or 2.
2. Type T)ransfer protocol <enter>.
3. Select which file transfer method you want - (X, Y, or Z protocol).
4. Type D)ownload *filename* <enter>. *Filename* is name it will be called on the BBS.
5. Select the receive file utility on your software package. If you are using a Procomm like software package, you press the "Page Down" key.

Electronic Mail

Lattice Semiconductor does support the use of "E Mail". Communications regarding ispLSI or pLSI products can be sent to "applications@lattice.com". Please include details as well as a voice telephone number.

Using the Hillsboro BBS

The Hillsboro BBS is accessible by any user with a modem and communication package. You can use the Hillsboro BBS to:

1. Transfer designs to and from Lattice Applications Engineers
2. Access the latest utilities
3. Join conferences to share and exchange information with Lattice Applications Engineers and other users
4. Send mail to and from Lattice Applications Engineers

Telephone Number and Communication Software Setup

The telephone number for the Hillsboro BBS is (503) 693-0215. The BBS supports modem speeds from 300-9600 Baud, and supports the typical default communication parameters of eight data bits, one stop bit, and no parity (8-N-1).

If You Are A New User

If you have not used the BBS before, the system will ask you a short set of questions. These questions are used to maintain statistics about our callers, and will not take long to answer.

You will be asked for a user name and password. For the user name, simply enter your name. You will also be prompted to enter a password. It is important to remember the password you enter. You will need it whenever you log on to the system, and if you forget it, you may have to have your account information deleted, and you will have to log on again as a new user.

After you complete the questionnaire, the system will display the main menu. From this point on, you will see the main menu after you log on and give the system your name and password.

The Main Menu

The Main Menu is the top level menu, meaning that you can access all other menus from this point. The options you are most likely to use are:

- | | |
|----|-----------------|
| f- | file menu |
| j- | join conference |
| m- | message menu |

Use the file menu when you want to upload or download files. Use the join conference menu when you want to join a conference on a particular topic. Use the message

Figure 4. Hillsboro BBS Main Menu

MAIN MENU:	
[M].....Message menu	[F].....File menu
[C]....Comments to the sysop	[P].....Page the sysop
[I]...Initial welcome screen	[Y].....Your settings
[G].....Goodbye & Logoff	[H].....Help level
[?].....Command help	[J].....Join conference
Conf: "[0] - Lattice Technical Support", time on 0, with 60 remaining.	
MAIN MENU: [M F C P I Y G H ? J] ? []	

Lattice Bulletin Board Systems

menu when you want to leave a message, either as part of a conference, or to a specific individual.

The File Menu

When you choose the File menu from the Main menu, you will be presented with a list of options for uploading, listing, or downloading files on the BBS.

If you need to download a file, and you know the name of the file you want to download, choose option D and download the file. You don't have to be in a particular conference to download the file.

If you don't know the exact name of the file, then you can choose one of the options in figure 6 to locate the file.

Figure 7 is an example session where files in a particular area are listed. In this example, when "L" was entered for the file area, a list of all the file areas was displayed. From this list of areas, you can choose the specific area you are interested in listing.

Transferring Files

Once you identify the file that you want to download, choose the D option from the Download menu. You will be prompted for the file name. Alternatively, you can initiate a download after listing the files in an area. Note that one of the options in the menu listed above is D for download.

File Protocols

The BBS supports a number of different file protocols for downloading and uploading files. Which one to choose depends on your communication software. Xmodem is one of the most popular protocols, and your communication software is likely to support it.

You can display the file transfer protocol you've chosen by selecting the Y option from the main menu (see figure 8).

When you choose the Y option, all configuration parameters are displayed. Number 14 is the file transfer protocol. By entering 14 as the setting to change, you can change to a different file transfer protocol. Note that you can choose to select the file transfer protocol each time you start a download by selecting S as the default protocol option.

Conferences

The Hillsboro BBS also provides a conference facility for you to share information with Lattice Application Engineers, and other users of Lattice Products. Conferences are simply a way to organize messages left by users so that they are grouped by a common subject. When you join a conference, messages that you read or leave will then be left in that conference area.

Figure 5. Hillsboro BBS File Menu

MAIN MENU: [M F C P I Y G H ? J] ? [F]	
FILE MENU:	
[D].....Download a file(s)	[U].....Upload a file(s)
[L].....List available files	[Q].....Quit to main menu
[N].....New files since [N]	[I].....Information on a file
[T].....Text search	[F].....File transfer info
[G].....Goodbye & logoff	[H].....Help level
[?].....Command help	[M].....Message menu
[V]...View a compressed file	[R].....Read a text file
[J].....Join conference	[E].....Edit marked list
Conf: "[0] - Lattice Technical Support", time on 2, with 58 remaining.	
FILE MENU: [D U L Q N I T F G H ? M V R J E] ? []	

Figure 6. Finding a File Without Knowing the Specific Name

If You Want to List Files	Then Choose Option	And Enter
Uploaded after a certain date	N	The starting date for the search
Containing a specific word in their name or description	T	The word to search for
Within an area category	L	The area to list

Lattice Bulletin Board Systems

You can join a conference from the Main menu by entering the J command (see figure 9). After entering this command, you can either enter the number of a conference you want to join, or enter an L to list the available conferences.

Once the conference is joined, you can enter the Message menu from the Main menu (see figure 10), and read new messages in the conference by entering the R

command, or you can enter the S command to scan for new messages.

The Scan command can be an easy way to locate topics of interest. The Scan menu will list a variety of options to search through messages. For example, you can specify a word to search for anywhere in the body of a message by selecting the B option. Any messages that contain this text will be displayed (see figure 11).

Figure 7. Example Session Showing Files Listed in a Particular Area

```
FILE MENU: [D U L Q N I T F G H ? M V R J E] ? [L]
Areas (1..8) [# , #-#], [A]ll, [L]ist, [S]D[F], [H]elp)? 1

Scanning file area - GAL Applications Info
[ 1] 20VP8.ABL          988    01/13/93 | ABEL example of setting output type
      DwnLds: 26      DL Time 00:00:05 | for GAL20VP8
[ 2] CHKSUM.EXE        4,992    11/13/91 | Simple JEDEC Fuse Checksum Utility
      DwnLds: 73      DL Time 00:00:26 | *Info*
[ 3] PALTOGAL.EXE     33,012    12/09/92 | Ver. 3.12, util to convert PAL JEDEC
      DwnLds: 277     DL Time 00:02:51 | files to GAL files
[ 4] PHY              836    01/13/93 | ABEL example of setting output type on
      DwnLds: 27      DL Time 00:00:04 | GAL16VP8
[ 5] XSUM.EXE         14,069    01/18/89 | Simple JEDEC Transmission Calculation
      DwnLds: 52      DL Time 00:01:13 | Utility *Info*

End of list
-Pause- [C]ont, [H]elp, [N]onstop, [M]ark, [D]wnld, [I]nfo, [V]iew, [S]top? [C]
```

Figure 8. Selecting the Y Option from the Main Menu

```
MAIN MENU: [M F C P I Y G H ? J] ? [Y]

Present setting for : NEW USER

[ 1] Password          : *****      Msgs written : 0
[ 2] Computer type     : 8088 based syst No. of calls : 3
[ 3] Phone number      :               High message : 0
[ 4] Birth date        : / /           User since   : 04/01/94
[ 5] Screen length     : 23            Last call    : 04/05/94 11:12am
[ 6] Color menus       : NO             Last new files: 01/01/80 12:00am
[ 7] Erase prompt      : NO             Downloads   : 0 Files, OK
[ 8] Hot keys          : NO             Uploads     : 0 Files, OK
[ 9] Quote on Reply    : NO             Security level: NEWUSER
[10] Msg Clear Screen : NO             Acct balance: 0
[11] Default editor    : No default     Netmail balance: 0
[12] File display mode: Double line
[13] Help level        : Novice
[14] Default protocol  : All
[15] Calling from      :
[16] Chat status       : Unavailable

Setting to change [1..16], [H]elp ? [ ]
```

Lattice Bulletin Board Systems

Figure 9. Joining a Conference

```
MAIN MENU:
[M].....Message menu      [F].....File menu
[C]....Comments to the sysop [P].....Page the sysop
[I]...Initial welcome screen [Y].....Your settings
[G].....Goodbye & Logoff    [H].....Help level
[?].....Command help        [J].....Join conference

Conf: "[0] - Lattice Technical Support", time on 19, with 40 remaining.

MAIN MENU: [M F C P I Y G H ? J] ? [J]

Join conference [0-3], [L]ist, [H]elp? [L ]
Conferences available:

    0) Lattice Technical Support    1) Private E-Mail
    3) Utilities
Join conference [0-3], [L]ist, [H]elp? [ ]
```

Figure 10. Message Menu

```
MESSAGE MENU:
[Q]....Quit to the main menu [J].....Join conference
[R].....Read messages        [S].....Scan messages
[E].....Enter a new message  [K].....Kill a message
[G].....Goodbye & logoff     [H].....Help level
[?].....Command help        [F].....File menu

Conf: "[0] - Lattice Technical Support", time on 14, with 44 remaining.

MESSAGE MENU: [Q J R S E K G H ? F] ? [ ]
```

Figure 11. The Scan Command

```
MESSAGE MENU: [Q J R S E K G H ? F] ? [S]

[F]rom      : <ALL>
[T]o        : <ALL>
[S]ubject   : <ALL>
Msg [B]ody  : <ALL>
[N]umber    : <ALL>
[D]irection : Forward
[C]onference : Current

Search command [F T U N D B C], [H]elp, [S]tart, [ENTER] to Quit? [B]
Search text? [paltogal ]

[F]rom      : <ALL>
[T]o        : <ALL>
[S]ubject   : <ALL>
Msg [B]ody  : PALTOGAL
[N]umber    : <ALL>
[D]irection : Forward
[C]onference : Current

Search command [F T U N D B C], [H]elp, [S]tart, [ENTER] to Quit? [ ]
```

Cost of Ownership: An Overview

Everybody's talking about it, but what is it really?

Simply stated, Cost of Ownership is the notion that the total cost of a particular system component (in this case a PLD), is the summation of all costs incurred throughout the "life" of that component which can be directly attributed to that component. In many cases, the initial purchase price of a component, the standard by which most engineers and purchasing agents judge component costs, is only a small fraction of the total costs that will be incurred by the component during its "life".

In fact, the initial purchase price of a PLD should be viewed only as the starting point of the Total Cost of Ownership "Life Cycle". As the PLD undergoes early stages of processing, it continues to incur costs. Costs associated with incoming inspection/rejection, inventory management, programming, programming yield loss, labeling and device handling are but a few of the costs incurred just getting the PLD ready for board assembly.

The next phase of the PLD's Cost of Ownership Life Cycle can be described as the board/system assembly and test process steps. At this point, the stakes get higher. PLDs which do not smoothly integrate into the board/system process flow, or worse yet, fail after board/system assembly are very expensive devices indeed.

A system shipped to an end customer has now entered the final phase of the PLD Life Cycle. The issue here is simply reliable system performance. PLDs which consistently meet and exceed system performance standards

and do not fail in the Field make no further contributions to the Total Cost of Ownership equation. However, field failures can be catastrophic and immeasurable, not only in terms of actual costs of field repairs but in terms of damage to customer "Good Will".

For Lattice PLDs, fundamental product and process characteristics such as the generic (flexible) architecture, EECMOS reprogrammability and in-system reprogrammability, 100% tested E² cells (Zero fall-out) and proven industry leading quality and reliability all combine to give the lowest Cost of Ownership of any PLD.

Lattice recognizes that the actual cost savings obtained by designing with Lattice's GAL, ispLSI and pLSI devices will vary from application to application. The total cost savings realized is a function of how broadly the ISP device capabilities are implemented throughout the product development and manufacturing cycles (Design, Programming, Test, Field Upgrades, etc.).

The following three papers examine the "Cost of Ownership" concept from three different perspectives.

1. *Hidden Costs in PLD Usage* — "Must reading" for any designers who are still using bipolar PAL devices. This paper describes the Cost of Ownership benefits of the Lattice GAL family vs bipolar PALs.

2. *ISP: Winning at the Bottom Line* — Anyone interested in enhanced design functionality, rapid time-to-market, simplified manufacturing and easy field upgrades will be interested in this article.

3. *Gate Array and High-Density PLD Cost Analysis* — Provides a detailed cost model for Gate Arrays as well as the various FPGA technologies in the market today including SRAM, EPROM, Anti-fuse and EECMOS.

Cost of Ownership An Overview

and do not fall in the field make no further contribution to the Total Cost of Ownership equation. However, field failures can be catastrophic and irreparable, not only in terms of actual costs of field repairs but in terms of damage to customer "Good Will."

For lattice PLDs, fundamental product and process characteristics such as the generic (flexible) architecture, EECMS reprogrammability and in-system testability, 100% tested E² cells (Zero fail-out) and proven industry leading quality and reliability all combine to give the lowest Cost of Ownership of any PLD.

Lattice recognizes that the actual cost savings obtained by designing with Lattice's GAL, ispLSI and ispLSI devices will vary from application to application. The total cost savings realized is a function of how broadly the ISP device capabilities are implemented throughout the product development and manufacturing cycles (Design, Programming, Test, Field Upgrades, etc.).

The following three papers examine the "Cost of Ownership" concept from three different perspectives.

1. Hidden Costs in PLD Usage — "Must reading" for any designer who are still using bipolar PAL devices. This paper describes the Cost of Ownership benefits of the Lattice GAL family vs bipolar PALs.

2. ISP: Winning at the Bottom Line — Anyone interested in enhanced design functionality, rapid time-to-market, simplified manufacturing and easy field upgrades will be interested in this article.

3. Gate Array and High-Density PLD Cost Analysis — Provides a detailed cost model for Gate Arrays as well as the various FPGA technologies in the market today including SRAM, EPROM, Antifuse and EECMS.

Everybody's talking about it, but what is it really?

Simply stated, Cost of Ownership is the notion that the total cost of a particular system component (in this case a PLD) is the summation of all costs incurred throughout the "life" of that component which can be directly attributed to that component. In many cases, the initial purchase price of a component, the standard by which most engineers and purchasing agents judge component costs, is only a small fraction of the total costs that will be incurred by the component during its "life."

In fact, the initial purchase price of a PLD should be viewed only as the starting point of the Total Cost of Ownership "Life Cycle." As the PLD undergoes early stages of processing, it continues to incur costs. Costs associated with incoming inspection, inventory management, programming, programming yield loss, labeling and device handling are but a few of the costs incurred just getting the PLD ready for board assembly.

The next phase of the PLD's Cost of Ownership Life Cycle can be described as the board/system assembly and test process steps. At this point, the stakes get higher. PLDs which do not smoothly integrate into the board/system process flow, or worse yet, fail after board/system assembly are very expensive devices indeed.

A system shipped to an end customer has now entered the final phase of the PLD Life Cycle. The issue here is simply reliable system performance. PLDs which consistently meet and exceed system performance standards

Hidden Costs in PLD Usage

While the purchase price of a programmable logic device is an important consideration in identifying the most cost-effective solution for a system design, it is clearly not the only criterion. Hidden costs attributable to product testing, yield fallout, inventory management, and other factors can dramatically impact the final cost of using a PLD.

This brief investigates the overhead associated with PLD usage and the advantages of testable and reprogrammable E²CMOS GAL devices over one-time-programmable PLDs.

The GAL family of programmable logic devices is manufactured on a state-of-the-art E²CMOS process that not only provides a better speed-power product than the best bipolar devices, but offers an advantage unique among PLD manufacturers: guaranteed programming and post-programming yields of 100%.

The 100%-yield guarantee is the culmination of years of Lattice Semiconductor's circuit-design and manufacturing experience applied to the GAL device. The only way to be able to make this 100% yield statement—and to supply product that actually meets the 100% criterion—is to fully test all functions of the device, prior to shipment.

The electrically erasable (EE) matrix, unlike previous PLD matrix technologies (bipolar fuse-link and UV-erasable PROM), permits full testing of the programmability and reprogrammability of each and every matrix cell. The ability to pattern the actual matrix is extremely significant, since it also allows Lattice to test the functionality of each of the Macrocell logic blocks, under various worst-case configurations. This test approach is referred to at Lattice as 'Actual Test'. Unlike other PLD manufacturers' approaches, which include imprecise correlations, simulations, test rows, and phantom arrays, Actual Test conclusively verifies AC and DC performance of every cell in every GAL device.

Eliminates Incoming QA

A consequence of Actual Test is that GAL devices do not require the typical incoming Quality Assurance testing that traditional fuse-link bipolar PLDs require. As such, the cost savings of using GAL devices begins the moment the parts arrive, since the average cost of an incoming QA operation—hardware, software development and maintenance, and handling—is approximately 7% of the raw device cost. Moreover, GAL devices become the optimal choice for implementation of Just-In-Time or Dock-To-Stock programs, since they eliminate the expense and time required by the incoming inspection process.

Still, a number of users require that all devices undergo incoming QA. In those cases, the use of GAL devices still simplifies the issue. A single generic test program can be used to test all configurations of the E²CMOS-based GAL device. The expense of generating and maintaining a test program for every architecture (16L8, 16R4, 10P8, and so on) is eliminated with Generic Array Logic.

Since the QA test for fuse-link PLDs, by its nature, requires the destructive patterning of the fuse array, QA testing of bipolar PAL devices can only be done through a sample plan. At best, a sample plan can provide a crude estimate of fuse-link yield loss; moreover, sampled devices cannot be erased and must be subsequently thrown away. GAL devices, utilizing the E²CMOS process can be patterned and erased at will, allowing 100% QA of all specifications and configurations. And, the devices can, of course, be erased to allow full reuse of the sample units in manufacturing.

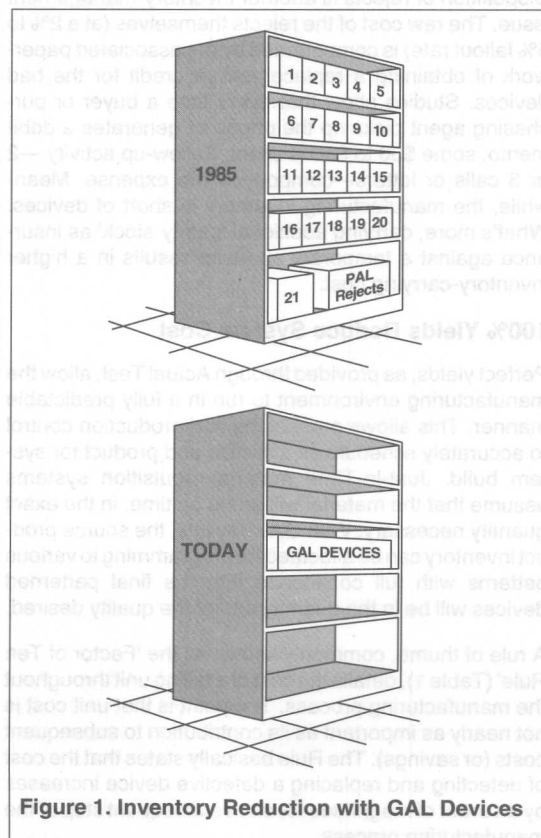


Figure 1. Inventory Reduction with GAL Devices

Hidden Costs in PLD Usage

Simplified Inventory Management

The generic architecture and high performance of the GAL devices allow five basic devices—the GAL16V8, GAL20V8, GAL22V10, GAL20RA10 and GAL20XV10—to directly replace approximately 98% of bipolar PLD device types currently available. The obvious benefit of using GAL devices is a substantial reduction in the number of part types that need be stocked (Figure 1).

Inventory management of dozens of speed-power options and device architectures is a painful process. The ideal cost of managing a device inventory adds some 2% of direct overhead; the real cost can be significantly greater, due to the risk that a shortage, 'outage,' or obsolete stock condition will exist. Improper planning could result in a shut-down of the assembly line. The generic architecture allows the GAL device to serve as insurance whenever needed to meet an immediate short-fall. The yields, at 100%, allow full planning confidence that the problem is solved.

Disposition of rejects is another inventory-management issue. The raw cost of the rejects themselves (at a 2% to 5% fallout rate) is compounded by the associated paperwork of obtaining a replacement or credit for the bad devices. Studies show that every time a buyer or purchasing agent picks up the phone or generates a debit memo, some \$30 to \$50 is spent. Follow-up activity—2 or 3 calls or letters—compounds the expense. Meanwhile, the manufacturing inventory is short of devices. What's more, carrying additional 'safety-stock' as insurance against a temporary shortage results in a higher inventory-carrying cost.

100% Yields Reduce System Cost

Perfect yields, as provided through Actual Test, allow the manufacturing environment to run in a fully predictable manner. This allows purchasing and production control to accurately schedule all activities and product for system build. Just-In-Time material-requisition systems assume that the material will arrive on time, in the exact quantity necessary. With GAL devices, the source product inventory can be allocated for programming to various patterns with full confidence that the final patterned devices will be in the quantity and of the quality desired.

A rule of thumb, commonly known as the 'Factor of Ten Rule' (Table 1), details the cost of a failing unit throughout the manufacturing process. The point is that unit cost is not nearly as important as its contribution to subsequent costs (or savings). The Rule basically states that the cost of detecting and replacing a defective device increases by an order of magnitude for each subsequent step of the manufacturing process.

COST*	MULTIPLIER	OPERATION
\$ 5.00	X	Raw Cost of Device
\$ 50.00	10X	Cost of Detecting and Repairing a Board Failure
\$ 500.00	100X	Cost of Detecting and Repairing a System Failure
\$ 5,000.00+	1,000X	Cost of Repairing a Field Failure

Each successive operation results in 10 times the cost to detect the failing device. \$5.00 device cost assumed — use your actual cost and a 10x multiplier to obtain actual numbers.

Table 1: Factor of Ten Rule

It is extremely important to recognize that the additional difficulty and cost of using traditional PLDs has implications far beyond what the observed programming yield fallout portends. The hidden costs, time and expense aggravation of board failures (10x device cost to detect and repair), system failures (100x device cost), and the potential for field failures far outweigh the simple 2% to 5% yield losses observed on a programming fixture.

Figure 2 illustrates the differences between traditional PAL device yield loss and the 100% yields of the GAL devices. Notice that even operator errors and engineering pattern revisions are recoverable with GAL devices, which can be instantly erased and reprogrammed to the proper architecture and logic pattern.

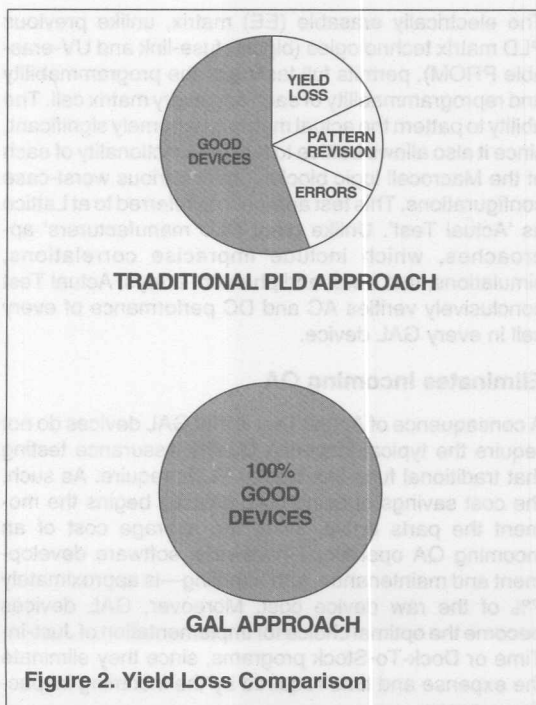


Figure 2. Yield Loss Comparison

Hidden Costs in PLD Usage

In a typical manufacturing environment, device programming hardware patterns the array, and assuming the engineer has provided test vectors, the hardware performs a basic (slow) functional test of the device. Yield losses at these two operations average 2% to 5% and 1% to 2%, respectively.

What is not tested adequately at the PAL programming operation is the effect of partially programmed fuses that result in degraded AC performance or marginal reliability of the device. These failures are caught at board test and/or after board burn-in. Typical bipolar functional and AC parametric failure rates range between 0.5% and 2% for all manufacturers of fuse-link PAL devices. Even if one assumes the minimum failure rate of 0.5%, the system failure rates are still greatly magnified.

Two mechanisms are used to detect the failures of PAL devices: board test and system test. Using the 'Factor of Ten Rule' and assuming that board test fully screens bad devices (AC fallout), if a conservative device failure rate of 0.5% were observed, the actual parts cost would be:

$$\begin{aligned} \text{Acost} &= \text{Pcost} + (\text{Pcost} * 10 * 0.5\%) \\ &= \text{Pcost} + \text{Pcost} * 0.05 \\ &= 1.05 * \text{Pcost} \end{aligned}$$

Performing the screening at the system level, under the same scenario, makes a dramatic difference in the cost of the device:

$$\begin{aligned} \text{Acost} &= \text{Pcost} + (\text{Pcost} * 100 * 0.5\%) \\ &= \text{Pcost} + \text{Pcost} * 0.5 \\ &= 1.5 * \text{Pcost} \end{aligned}$$

These two different cost factors were determined using the conservative failure rate of 0.5%. Using the GAL

device, with its 0% failure rate, provides instead a cost factor of 1; i.e., no additional cost burden is generated.

The problem caused by PLD failures obviously grows in proportion to the number of devices in a system, since the probability of a failure among a group of PAL devices is higher than that for a single device. Figure 3 plots the probability of a board or system not working, as a function of the number of devices per system, for a variety of device failure rates.

For example, at a unit failure rate of 1.0%, a system incorporating 30 PAL devices will exhibit a 25% failure rate. That means that 1 out of every 4 systems will have to be reworked, at tremendous cost. The replacement of an average 0.5% of the units in a system results in an actual 8% adder to the hidden device cost.

The difficulty in replacing board failures is compounded by the removal of soldered units. It is quite easy to destroy a board with the removal and replacement of a defective device.

Systems that fail in the field are not only the most costly in terms of dollars and cents, but in customer relations, as well. They require responding rapidly and performing repairs in a less-than-ideal environment, without the complete tools and supplies available at the factory. Field failures will always occur to some degree. However, the use of GAL devices can help reduce field repair costs when they do occur—even if the failing device is a traditional bipolar PLD—since the generic, erasable nature of GAL devices allows a minimum of field inventory to be carried, to debug system failure problems caused by other devices. The panel on the next page provides guidelines for calculating PLD usage costs.

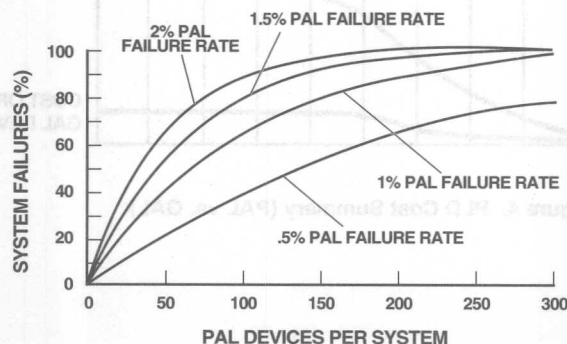


Figure 3. Probability of System Failures Using Bipolar PLDs

Hidden Costs in PLD Usage

PLD Cost Analysis

The cost of using a PLD goes well beyond simply the raw device cost. Programming and vector-test yields are obvious contributors to higher unit cost. The less-obvious and hidden costs tend to be much more difficult to identify and quantify.

The purpose of the costing example is to provide the basis for your own cost analysis, using your own overhead and yield numbers. Estimates for reasonable ranges of the cost contributors are shown as a guide to using your own numbers.

The explanation for each of the contributors to the device cost multiplier follow the figure. These cost multipliers include the overhead for each operation, and as a result, are higher than the observed costs.

The example shown is based on actual data from a 100,000-piece-per-year user of traditional bipolar PLDs. The environment is a typical, high-volume, quality-controlled one. The GAL device checks in at 1.09 times the normalized cost, while the actual cost of using the bipolar PLD is 1.66 times—almost 40% higher (figure 4).

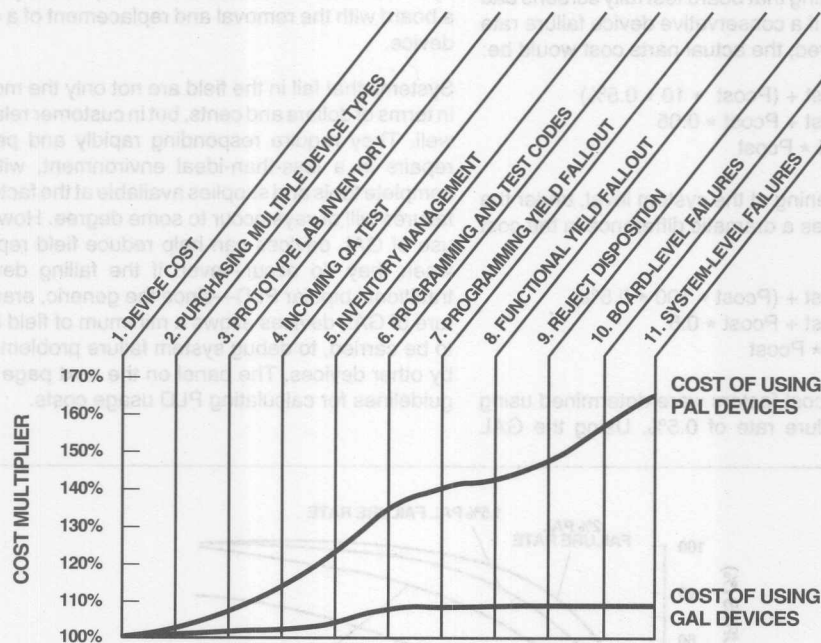


Figure 4. PLD Cost Summary (PAL vs. GAL)

1 **Device Cost** is normalized to unity so that the raw purchase price has no bearing on the other cost factors.

2 Purchasing Multiple Device Types

instead of the single GAL device adds to overhead in the purchasing and receiving departments. This contributes approximately 2% as the availability, quality, and quantity issues are resolved with each order. The GAL approach reduces this number to 1.25% with inventory simplification.

3 **Prototype Lab Inventory** and usage typically adds 5% to maintain experimentation stock of multiple device types for board debug. The GAL device multiplier is 1%, since the device can be reused over and over again.

4 Incoming QA Test

and programs cost more than may be immediately apparent, with a 7% adder. The generation and maintenance of the software and hardware for the dozens of bipolar devices is considerably more expensive than the single GAL device software required. No sample-program waste is induced. Only the aspects of handling are required for GAL devices, resulting in a reduction to 1% (or 0%, if you eliminate the incoming QA operation entirely).

5 Inventory Management includes shelf space, safety stock, depreciation, obsolete stock write-off and personnel to maintain adequate control of the units. A typical overhead is 10%. The simplified GAL operation involves no safety or obsolete stock and a minimum of device types, adding a maximum 2% to 3% to overhead.

6 **Programming and Test** includes all handling and hardware expenses. Inventory issuance, counting and returns, handling during the program/test operation, labels, and paperwork contribute to a 12% multiplier. The 100% yielding, generic GAL approach reduces the problem to 4%.

7 Programming Yield Fallout is directly observed as bad units. A typical bipolar range is 1% to 4%. GAL devices have 0% yield fallout—guaranteed.

8 Functional Yield Fallout

is detected by the device programmer immediately after programming, through the use of test vectors, and can average 1% to 3%. GAL devices guarantee 0% functional fallout. It should be noted that using test vectors does not screen out inadequate for AC performance, which will be manifested as a board failure.

9 Reject Disposition overhead runs 5% to obtain replacements and credits for fuse-link devices. Zero rejects with GAL devices eliminates costs associated with reject disposition. Notice that the cumulative multiplier for only the program/test/reject of fuse-link devices is 1.10, compared with GAL devices' 1.00 multiplier.

10 Board-Level Failures

are typically where AC failures are detected. The 'Factor of Ten Rule' exacerbates the impact of the observed 1% to 4% fallout to an overall cost impact of 7% to 10%. GAL devices exhibit no board-level fallout (and therefore no cost impact). Board throughput is also a major cost contributor, with typical reworks of 20% to 30% a consequence of PAL quality levels.

11 **System-Level Failures** add 8% to 15% to the PLD cost, taking into consideration a 100x 'Factor of Ten Rule' multiple. GAL devices again provide 100% yields, and therefore exhibit no system-level-failure cost impact

ISP: Winning at the Bottom Line

ISP (In-System Programmability) offers its users the opportunity to increase product value and reduce bottom-line cost simultaneously.

Added Product Value

ISP allows the system manufacturer to build in added product value through the addition of unique product capabilities. Taking advantage of in-system reconfigurability will differentiate and enhance the market positioning of an ISP-based product, ultimately resulting in higher sales and expanding market share.

Lowest Cost

Total cost of ownership goes far beyond the initial purchase price of components. ISP reduces costs throughout the entire life cycle of a product: product development, manufacturing, test, field maintenance and upgrades. This results in a cost savings of one third or more when compared to typical non-ISP devices.

Product Development Cost

ISP can reduce design development expense by allowing logic iterations to occur in minutes instead of hours as is common with other high-density PLDs. Also, ISP eliminates last-minute component changes and printed circuit board redesigns encountered during system debug. These features typically result in a greater than 50% reduction in design cycle time, engineering expense and time to market.

Programming Cost

ISP reduces programming cost through guaranteed 100% programming yields and the elimination of the need for a stand-alone device programmer and programming cycle. Devices are mounted onto the PCB directly from inventory. The reduced handling of the components translates into lower cost and higher quality with no bent leads or incorrectly patterned devices.

Functional Yield Cost

Higher product quality and reliability achieved through the superior test coverage of ISP and E²CMOS eliminates functional yield loss. Special test logic can be programmed temporarily into the hardware to facilitate exhaustive product and board testing. The elimination of defects at an early stage of board check-out reduces more expensive system-level failures later in the final manufacturing process.

Board Rework Cost

The superior test coverage and flexibility of in-system reprogrammability eliminates board rework cost. A design fault found in any component can potentially be corrected with a logic change in the ISP devices. No physical rework of the PCB would be required.

System Upgrades and Repair Cost

Lasting benefits from the use of ISP can be realized even after systems are shipped. In-system reprogramming can reduce field maintenance costs through enhanced field diagnostic capability, less costly product feature upgrades and simpler maintenance procedures. Training, documentation and on-going support can also be simplified by using the ISP approach to build in maintainability.

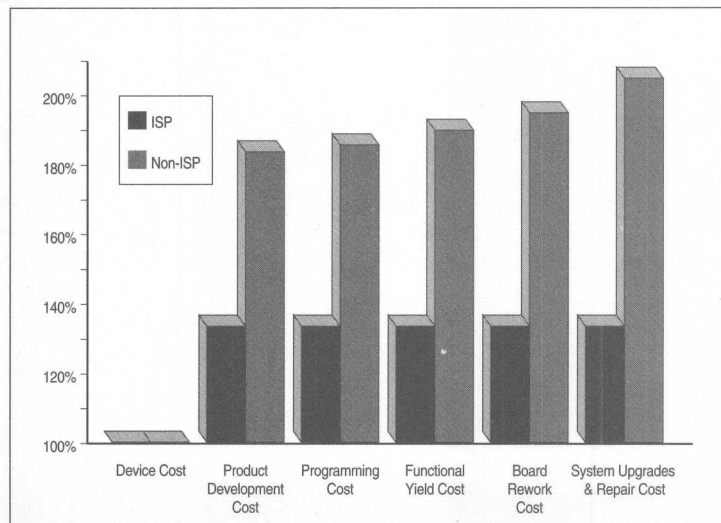


Figure 1. Cost of Ownership Comparison

Notes

ISP: Winning the Bottom Line

Functional Yield Cost

Higher product quality and reliability achieved through the superior test coverage of ISP and E²CMOS eliminates functional yield loss. Special test logic can be programmed temporarily into the hardware to facilitate extensive product and board testing. The elimination of defects at an early stage of board check-out reduces more expensive system-level failures later in the final manufacturing process.

Board Rework Cost

The superior test coverage and flexibility of in-system reprogrammability eliminates board rework cost. A design fault found in any component can potentially be corrected with a logic change in the ISP device. No physical rework of the PCB would be required.

System Upgrades and Repair Cost

Lasting benefits from the use of ISP can be realized even after systems are shipped. In-system reprogramming can reduce field maintenance costs through enhanced field diagnostic capability, less costly product feature upgrades and simpler maintenance procedures. Training, documentation and on-going support can also be simplified by using the ISP approach to build in maintainability.

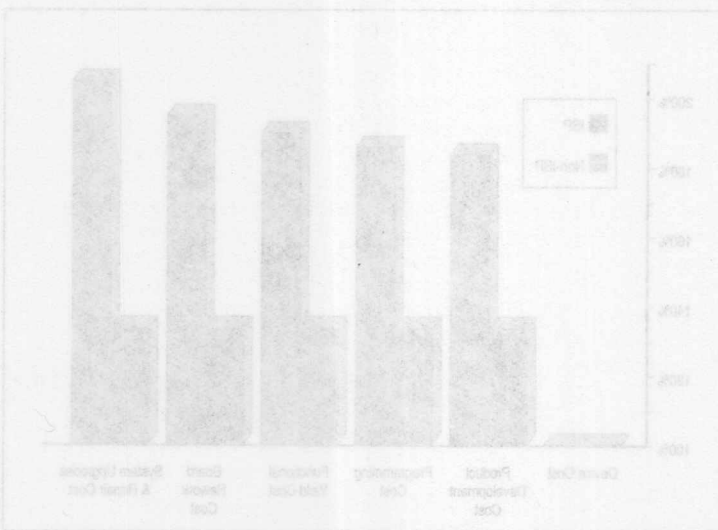


Figure 1: Cost of Ownership Comparison

ISP (In-System Programmability) offers its users the opportunity to increase product value and reduce total-line cost simultaneously.

Added Product Value

ISP allows the system manufacturer to build in added product value through the addition of unique product capabilities. Taking advantage of in-system reprogrammability will differentiate and enhance the market positioning of an ISP-based product, ultimately resulting in higher sales and expanding market share.

Lowest Cost

Total cost of ownership goes far beyond the initial purchase price of components. ISP reduces costs throughout the entire life cycle of a product: product development, manufacturing, test, field maintenance and upgrades. This results in a cost savings of one third or more when compared to typical non-ISP devices.

Product Development Cost

ISP can reduce design development expense by allowing logic iterations to occur in minutes instead of hours as is common with other high-density PLDs. Also, ISP eliminates last-minute component changes and pinched circuit board redesigns encountered during system debugging. These features typically result in a greater than 50% reduction in design cycle time, engineering expense and time to market.

Programming Cost

ISP reduces programming cost through guaranteed 100% programming yields and the elimination of the need for a stand-alone device programmer and programming cycle. Devices are mounted onto the PCB directly from inventory. The reduced handling of the components translates into lower cost and higher quality with no burnt leads or incorrectly patterned devices.

Gate Array and High Density PLD Cost Analysis

Introduction

When analyzing the total cost of a component, there are different factors which must be considered. This article explains these factors from the initial design considerations to the final system manufacturing using a typical example. Four different technologies (E²CMOS standard and in-system programmable (ISP) versions, UVC MOS, Anti-Fuse & SRAM) for the high density PLDs were used to make the comparison with the custom gate array to show the advantages and disadvantages of the different technologies. There is more to the cost of a component than the actual unit price. The attached table was developed to itemize all the "hidden" costs that are associated with selecting a component from the beginning of the design to the system manufacturing stage. These figures are based on the typical prices that were available at the time of the table generation. Although absolute values can always be challenged, relative values really tell the story on device cost.

Fixed Costs

In the beginning of the design cycle a project manager has to consider the fixed costs that are associated with the use of a particular component. Within these fixed costs are Non-Recurring Engineering (NRE) costs such as masks cost for the gate array, engineering time for the design, and software and hardware tools that are needed throughout the design cycle.

It is important to note that the up front fixed costs associated with the gate arrays are significantly higher than the high density PLD's because of the nature in which gate arrays are developed. The most significant contributor of this up front fixed cost is the cost of the masks. As volumes get higher, amortization of these fixed costs over the larger volumes makes the cost of the device lower. Also important to the gate arrays during the initial stages of the development is the simulation efforts. In order to save future costs due to changes made after the silicon has been developed, a thorough check of the design for accuracy during design stage is necessary. The associated cost for simulation is also included in the fixed costs for the gate array. For the high density PLD these fixed costs are minimal. Software and hardware tools needed for the high density PLD design development cost less compared to the gate array tools due to the fact that the PLD tools only require the translation of logic implementation and are isolated from the chip level design complications. Since the design turn times are

longer for the gate arrays, any design change will require a longer time which translates into opportunity costs associated with not meeting the time-to-market schedules. All these items are summarized under the fixed costs and opportunity cost sections of the table.

Variable Costs

Once the design is complete, the product development cycle shifts from the design stage to the manufacturing and testing stages. The costs associated with these later product development stages are itemized under the variable costs section of the table. When considering the cost of a component, most associate the cost with the actual purchase price of the unit. As can be seen from the table the purchase price is only a small part of the total development cost. Inventory management, programming yield, functional test yield, board failures/rework costs, and system failure/rework costs are all part of the costs associated with the variable costs.

Reusability of a component is driven by the technology used in a component which significantly affects most of the line items in the variable cost table. Of the technologies in question, gate arrays and anti-fuse are the only two that do not provide this capability. All reusable components have some way of being reprogrammed. The method of reprogramming defines the testability during IC manufacturing. Of the technologies E²CMOS and the SRAM based components offer the best testability due to the very short reprogramming times. With improved testability of the components, manufacturers are able to guarantee 100% programming and 100% functional yields. These guarantees translate to lower variable costs during programming of the devices and manufacturing of the systems as reflected in the table. As an added feature, ispE²CMOS technology enables the user to program the device in-system so that the cost associated with any design changes on the component are accomplished transparently with no board rework costs. This capability is highlighted by not having the board and system level repair cost factors.

Summary

The attached table is generated with the following assumptions:

- 1) Approximately 8,000 gate complexity per component.
- 2) 100 boards with 3 sockets each.
- 3) 1 I.C. design iteration.

Gate Array and High Density PLD Cost Analysis

- 4) Programming and reliability consideration of the SRAM based FPGA does not take into the consideration the PROM that must be used to store the configuration.

In summary, there is definitely a cost crossover between the high density PLD and the gate array. In this particular example, the crossover volume is approximately 15K

units. The in-system programming capability the E²CMOS technology provides the most straight-forward and flexible way of implementing design changes in the shortest product development cycle. Among the high density PLD's the E²CMOS technology and the SRAM technology are the two technologies that gives the lowest overall costs.

Table 1. High Density Cost Analysis

	Gate Arrays	ispE ² CMOS	E ² CMOS	SRAM	UVC MOS	Anti-Fuse
FIXED COSTS						
NRE Charges (masks)	\$10,000	\$0	\$0	\$0	\$0	\$0
Design						
Hardware Tools	\$15,000	\$5,000	\$5,000	\$5,000	\$5,000	\$5,000
Software & Development Tools	\$10,000	\$1,390	\$1,390	\$5,000	\$9,950	\$9,690
Engineering Time (Man Weeks)	4	2	2	2	2	2
Simulation						
Tools NRE	\$5,000	\$0	\$0	\$0	\$0	\$0
Engineering Time (Man Weeks)	4	0	0	0	0	0
Device Test Program Generation (Man Weeks)	2	0	0	0	0	0
Design Change Iterations	1	1	1	1	1	1
NRE Cost / Iteration	\$5,000	\$0	\$0	\$0	\$0	\$100
Cost of Design Iteration	\$5,000	\$0	\$0	\$0	\$0	\$100
Engineering Time (Man Week)	2	0.5	0.5	0.5	0.5	0.5
Programming Equipment Cost	\$0	\$395	\$5,000	\$2,000	\$5,500	\$5,000
Total Fixed Cost	\$69,000	\$11,785	\$16,390	\$17,000	\$25,450	\$24,790
At \$2000/Man Week	\$2,000					
OPPORTUNITY COST						
TTM Cost/Day	\$25,000	\$25,000	\$25,000	\$25,000	\$25,000	\$25,000
Days for Silicon Turns	20	0.5	1	1	1	1
Total Opportunity Cost	\$500,000	\$12,500	\$25,000	\$25,000	\$25,000	\$25,000
VARIABLE COSTS						
Unit Price	\$50	\$125	\$100	\$100	\$100	\$100
Support Chip Cost	\$0	\$0	\$0	\$2	\$0	\$0
Number of Units / Board	3	3	3	3	3	3
Number of Boards	100	100	100	100	100	100
Total Device Cost	\$15,000	\$37,500	\$30,000	\$30,600	\$30,000	\$30,000
Inventory Cost / Item	\$1,000	\$1,000	\$1,000	\$1,000	\$1,000	\$1,000
Number of Items	3	1	1	1	1	1
Total Inventory Cost	\$3,000	\$1,000	\$1,000	\$1,000	\$1,000	\$1,000
%Programming Yield	n/a	100%	100%	100%	98%	95%
Administrative and Reprogramming Cost Factor	n/a	1.20	1.20	1.20	1.50	2.00
Cost of Programming Yield Lost	\$0	\$0	\$0	\$0	\$918	\$3,158
%Functional Yield	95%	100%	100%	100%	98%	95%
Administrative Cost Factor	1.20	1.20	1.20	1.20	1.50	2.00
Cost of Functional Yield Lost	\$947	\$0	\$0	\$0	\$918	\$3,158
%Board Failures	1.00%	0.00%	0.00%	0.00%	0.50%	1.00%
Board Rework Cost Factor	10	0	10	10	10	10
Cost of Board Failures	\$1,500	\$0	\$0	\$0	\$1,500	\$3,000
%System Failures	0.15%	0.00%	0.00%	0.00%	0.10%	0.15%
System Repair Cost Factor	100	0	100	100	100	100
Cost of System Failures	\$2,250	\$0	\$0	\$0	\$3,000	\$4,500
Total Variable Cost	\$22,697	\$38,500	\$31,000	\$31,600	\$37,337	\$44,816
Total Cost	\$591,697	\$62,785	\$72,390	\$73,600	\$87,787	\$94,606